Assembler

Grundlagen der PC-Programmierung

Von
Ernst-Wolfgang Dieterich

5., überarbeitete Auflage

Oldenbourg Verlag München Wien

Prof. Dr. Ernst-Wolfgang Dieterich, Fachhochschule Ulm, Fachbereich Elektrotechnik.

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

© 2005 Oldenbourg Wissenschaftsverlag GmbH
Rosenheimer Straße 145, D-81671 München
Telefon: (089) 45051-0
www.oldenbourg.de

Das Werk einschließlich aller Abbildungen ist urheberrechtlich geschützt. Jede Verwertung außerhalb der Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Bearbeitung in elektronischen Systemen.

Lektorat: Margit Roth
Herstellung: Anna Grosser
Umschlagkonzeption: Kraxenberger Kommunikationshaus, München
Gedruckt auf säure- und chlorfreiem Papier
Druck: Grafik + Druck, München
Bindung: R. Oldenbourg Graphische Betriebe Binderei GmbH

ISBN 3-486-20001-1

Inhaltsverzeichnis

1 Vorwort 1

2 Die Entwicklungsumgebung 5

2.1 Ein einfaches Beispiel ... 7

2.2 Assembler und Linker ... 8

2.3 Der Debugger ... 9

3 Allgemeiner Aufbau eines Assembler-Programms 13

3.1 Bezeichner ... 14

3.2 Befehle ... 16

3.3 Direktiven ... 17

3.4 Ein Standard-Rahmen für Assembler-Programme ... 18

3.5 Befehle und Direktiven des Beispielprogramms ... 19

3.6 Makro-Aufrufe ... 22
3.6.1 Ausgabe von Text ... 22
3.6.2 Einlesen von Text ... 25
3.6.3 Einlesen und Ausgeben von Zahlen ... 26
3.6.4 Zufallszahlen-Generator ... 29

3.7 Das Assembler-Listing ... 30
3.7.1 Der erweiterte Quell-Code ... 32
3.7.2 Die Symbol-Tabelle ... 34
3.7.3 Die Cross-Referenz-Liste ... 35
3.7.4 Steuerung der Listing-Ausgabe ... 36

4 Konstanten und Variablen 39

4.1 Ganze Zahlen ... 39

4.2 Zeichen und Zeichenketten ... 42

4.3 Vereinbarung von Variablen ... 43

4.4 Konstante Ausdrücke ... 47
4.4.1 Arithmetische Operatoren ... 49
4.4.2 Schiebeoperatoren ... 52

4.4.3 Logische Operatoren 53
4.4.4 Vergleichsoperatoren 54
4.5 Konstanten-Vereinbarung 55

5 Arithmetische Befehle 59

5.1 Die Register 59
5.1.1 Die allgemeinen Register 59
5.1.2 Die Index-Register 61
5.1.3 Die Zeiger-Register 61
5.1.4 Die Segment-Register 61
5.1.5 Der Befehlszeiger 62
5.1.6 Das Status-Register 62
5.2 Zuweisung, Addition und Subtraktion 65
5.3 Schiebebefehle 70
5.3.1 Die logischen Schiebebefehle 71
5.3.2 Die arithmetischen Schiebebefehle 73
5.3.3 Rotationsbefehle 74
5.4 Multiplikation 75
5.5 Division 80
5.6 Logische Befehle 83
5.7 Keller-Befehle 87

6 Kontrollstrukturen im Assembler 89

6.1 Die einseitige Verzweigung 90
6.2 Die zweiseitige Verzweigung 99
6.3 Bedingungsschleifen 100
6.4 Zählschleifen 107

7 Adressierungsarten 111

7.1 Die unmittelbare und die direkte Adressierung 111
7.2 Die Index-Adressierung 113
7.3 Die indirekte Adressierung 121
7.4 Die basis-indizierte Adressierung 123
7.5 Die index-indirekte Adressierung 128
7.6 Die Stringadressierung 131
7.6.1 Stringbefehle zum Datentransport 133
7.6.2 Stringbefehle zum Durchsuchen 139
7.6.3 Stringbefehle zum Vergleich von Speicherbereichen 143

8 Makros und bedingte Assemblierung 145

8.1 Makro-Definition und Makro-Aufruf ... 146

8.2 Blockwiederholungen ... 157

8.3 Bedingte Assemblierung ... 162
8.3.1 Allgemeine Bedingungsblöcke ... 164
8.3.2 Makro-spezifische Bedingungsblöcke ... 167
8.3.3 Geschachtelte Bedingungsblöcke ... 171

8.4 Geschachtelte Makros ... 172

8.5 Weitere Direktiven zur Steuerung der Listing-Ausgabe ... 176

9 Unterprogramme 179

9.1 Definition und Aufruf von Unterprogrammen ... 180

9.2 Parameter-Übergabe ... 182

9.3 Lokale Marken und lokale Variable ... 193

9.4 Ergebnisse aus Unterprogrammen ... 197

9.5 Spezielle Direktiven ... 200

9.6 Rekursive Unterprogramme ... 208

9.7 Aufruf von Interrupt-Routinen ... 216

9.8 Selbstgeschriebene Interrupt-Routinen ... 228

10 Segmente und Segment-Anweisungen 233

10.1 Die physikalische Speicheradresse ... 233

10.2 Die vereinfachten Segment-Anweisungen ... 239

10.3 Die Standard-Segment-Anweisungen ... 243

10.4 Die Segmente der vereinfachten Segment-Anweisungen ... 251

10.5 Kommandozeilen-Parameter ... 254

10.6 Der Bildschirm-Speicher ... 258

11 Modularisierung von Programmen 261

11.1 Sprachmittel des Assemblers zur Modularisierung ... 264

11.2 Der Binder ... 269

11.3 Die Bibliotheksverwaltung ... 275

12 Assembler und Hochsprachen 277

12.1 Die Schnittstelle zu C++ und C ... 280
12.1.1 Die Speichermodelle ... 280
12.1.2 C++ oder C ruft ein Assembler-Unterprogramm auf ... 281
12.1.3 Assembler ruft C++-Funktion auf ... 292
12.1.4 Verwendung gemeinsamer Daten ... 294
12.1.5 Ein Beispiel mit dem Speichermodell LARGE ... 297

12.2 Die Schnittstelle zu Pascal ... 299

12.3 Die Schnittstelle zu Turbo Pascal ... 299
12.3.1 Das Speichermodell von Turbo Pascal ... 300
12.3.2 Turbo Pascal ruft ein Assembler-Unterprogramm auf ... 302
12.3.3 Assembler ruft Turbo Pascal-Funktion auf ... 308
12.3.4 Verwendung gemeinsamer Daten ... 310
12.3.5 Das Speichermodell TPASCAL ... 311

13 Zusammengesetzte Datentypen 313

13.1 Strukturen ... 314

13.2 Variante Strukturen ... 319

13.3 Records ... 322

14 Andere Prozessoren 325

14.1 Einstellung des Prozessors ... 326

14.2 Die neuen und erweiterten Befehle des 80186 und 80286 ... 327

14.3 Der 32 Bit-Prozessor 80386 ... 331
14.3.1 Segment-Typen des 80386 ... 331
14.3.2 Erweiterte Verwendung der Register bei den Adressierungsarten ... 332
14.3.3 Neue Befehle des 80386 ... 333
14.3.4 Erweiterung vorhandener Befehle ... 337

Anhang 339

A Die verwendeten Makros ... 339

B ASCII-Tabelle ... 348

Literaturverzeichnis 350

Index 351

1 Vorwort

Vorwort zur 5. Auflage

Dieses Buch erscheint nun in seiner 5. Auflage. Für diese Auflage wurde der Text in das neue Oldenbourg-Format konvertiert, wobei der Text vom ventura-Format auf das TEX-Format überführt wurde. Der Autor hat sich bemüht – mit Unterstützung des Verlages – die gute Qualität des Textes mit möglichst wenigen Fehlern auf das neue Layout und das neue System zu übertragen. Der Autor dankt den TEX-Spezialisten des Verlags für die Unterstützung.

Vorwort zur 4. Auflage

Seit der 4. Auflage wird der Assembler für den Intel 8086-Prozessor unabhängig vom Hersteller behandelt.

Die Beispiele wurden mit den beiden Assemblern Turbo Assembler Version 3.1 und dem Microsoft Assembler MASM 5.0 getestet.

Die einzigen Teile des Buches, in denen sich der Autor auf eine spezielle Assembler-Version – nämlich den Turbo Assembler – bezieht, sind Kapitel 2 und Abschnitt 12.3: In Kapitel 2 wird anhand des Turbo Assemblers die Übersetzung und das Debuggen erläutert. In Abschnitt 12.3 wird die Schnittstelle zu Turbo Pascal behandelt, wobei auch spezielle Sprachkonstruktionen des Turbo Assemblers besprochen werden.

An wenigen Stellen unterscheiden sich Turbo Assembler und Microsoft Assembler; darauf wird explizit hingewiesen. Es wird folgende Schreibweise verwendet:

- TMASM: Turbo **und** Microsoft Assembler
- TASM: nur Turbo Assembler.

Die Beispiele stehen zum Download zur Verfügung (siehe letzte Seite). In der Beschreibung der Beispiele wird angegeben, welches Beispiel mit welchem Assembler übersetzt werden kann.

Vorwort zur 1. Auflage

Ist Assembler-Programmierung noch gefragt in einer Zeit, in der es immer bessere und benutzerfreundlichere Compiler für höhere Programmiersprachen wie Pascal, C oder

C++ gibt? Das ist wohl die zentrale Frage, die sich jeder schon oft gestellt hat, der ein Buch wie dieses aufschlägt. Ein klares Ja oder Nein kann auf diese Frage sicherlich nicht gegeben werden, so dass wir nur einige Vor- und Nachteile des Assemblers im Vergleich zu höheren Programmiersprachen aufzählen und die Zielsetzung dieses Buches vorstellen wollen.

Zuallererst muss klargestellt werden, dass es *den* Assembler überhaupt nicht gibt: Jeder Rechnertyp hat seinen eigenen Assembler, der den jeweiligen Befehlssatz und die spezifischen Eigenschaften abdeckt. Dies kann man als Nachteil oder als Vorteil des Assemblers beurteilen. Nachteilig ist sicherlich, dass man im Assembler keine portablen Programme schreiben kann; das sind Programme, die auf unterschiedlichen Rechnern laufen, ohne dass der Programm-Code verändert werden muss. Die Erstellung portabler Programme ist die Domäne der höheren Programmiersprachen. Dies hat allerdings zwangsläufig zur Folge, dass man in höheren Programmiersprachen keine speziellen Eigenschaften des verwendeten Rechners bis ins Letzte berücksichtigen kann. Die Ausnutzung dieser Spezial-Eigenschaften ergibt aber immer ein effektiveres Programm mit kürzerem Code und schnellerer Laufzeit. So werden auch heute noch viele Compiler – wie z.B. der Turbo Pascal-Compiler – in großen Teilen in Assembler geschrieben. Auch wenn es um zeitkritische Funktionen in einer Echtzeit-Umgebung geht, kann man meist nur mit einem Assembler-Programm die Zeitschranken einhalten. Eine weitere Stärke des Assemblers ist die hardware-nahe Programmierung, etwa die Erstellung von so genannten Treibern für Mess- und Steuergeräte, die über spezielle Schnittstellen an den Computer angeschlossen sind.

Die Fülle an verschiedenartigen Rechnern mit ihren eigenen Assemblern birgt die Gefahr in sich, dass man sehr viele verschiedene Assembler-Sprachen erlernen muss. Dieses Problem hat sich heute dadurch entschärft, dass sich gerade im PC-Bereich eine Standardisierung des Rechnertyps durchgesetzt hat. Im vorliegenden Buch werden der Turbo Assembler und der Microsoft Assembler behandelt, die die Maschinensprache der IBM-PC's und der Kompatiblen ist, die alle einen Prozessor der Familie Intel 80x86 enthalten.

Ein weiterer Vorteil der höheren Programmiersprachen ist der Zwang zur strukturierten Programmierung. Vor allem die Kontrollstrukturen zwingen den Programmierer, sauber strukturierte und leicht lesbare Programme zu schreiben. Der Assembler ist hier viel liberaler und lässt dem Programmierer alle Freiheiten, unstrukturierte und unlesbare Programme zu erstellen. Diese „Freiheit“ sollte man aber nicht auskosten. So wird im vorliegenden Buch an vielen Stellen immer wieder gezeigt, wie man auch im Assembler strukturierte Programme schreiben kann.

Zielsetzung des Buches

In den folgenden Kapiteln wird eine Einführung in das Programmieren im Assembler gegeben, wobei ganz bewusst auf die Hardware nur soweit eingegangen wird, wie dies zum Verständnis der einzelnen Befehle notwendig ist. Der Leser sollte bereits Erfahrung im Umgang mit seinem Computer besitzen und das Betriebssystem MS-DOS aus Benutzersicht kennen. Die Kenntnis einer höheren Programmiersprache ist vorteilhaft,

insbesondere wenn man die Schnittstellen zwischen Assembler und höheren Programmiersprachen nutzen will.

Das Buch wendet sich an Leser,

- die ihren Computer besser verstehen wollen,
- die Wirkungsweise der Compiler für höhere Programmiersprachen besser kennenlernen wollen, indem sie die Ausgabesprache der Compiler erlernen,
- ihre Programme, die in einer höheren Programmiersprache geschrieben sind, durch Einbinden von Assembler-Modulen verbessern wollen,
- Systemprogramme erstellen wollen und dabei maschinennahe oder zeitkritische Teile im Assembler realisieren müssen.

Der erlernte Stoff sollte unbedingt auf einem Rechner nachvollzogen werden. Prägnante kleine Beispiele sollen die einzelnen Lernschritte festigen. Da man im Assembler jedes Detail ausprogrammieren muss, ist der Weg bis zum ersten lauffähigen Assembler-Programm recht weit. Um diesen Weg abzukürzen – und dadurch auch den Einstieg zu erleichtern und den Spaß am Assembler-Programmieren zu steigern – sind wichtige Standard-Aufgaben wie die Ein- und Ausgabe in ein hochsprachen-ähnliches Gewand gekleidet: Es werden so genannte Makros verwendet. In der Mitte dieses Buches wird auf das Innenleben von Makros eingegangen. Spätestens dann kann man auch den Assembler-Code verstehen, der hinter diesen Makros steckt und in Anhang A abgedruckt ist. Wer sich das Abtippen dieser Makros und der Beispiele ersparen möchte, kann die Programmtexte herunterladen. Auf der letzten Seite ist beschrieben, wie man an die Dateien herankommt.

Gliederung

Im nächsten Kapitel wird an einem einfachen Beispiel dargestellt, wie man von der Assembler-Quelle zum ablauffähigen Programm kommt. Dabei werden kurz die beteiligten Werkzeuge aus der Entwicklungsumgebung vorgestellt, die man zusammen mit dem Turbo Assembler kauft. Dieses Kapitel kann und will kein Ersatz für die guten und ausführlichen Handbücher sein, die u.a. diese Werkzeuge und ihre Bedienung ausführlich beschreiben.

Im Kapitel 3 wird anhand des Einführungsbeispiels besprochen, wie Assembler-Programme aufgebaut sind. Ferner wird die Aufruf-Schnittstelle der verwendeten Makros beschrieben.

Die arithmetischen Befehle und ihre Operanden, nämlich Konstante, Variable und Register, werden in den Kapiteln 4 und 5 behandelt.

In Kapitel 6 werden die Kontrollstrukturen der höheren Programmiersprachen im Assembler nachgebildet. Die vielfältigen Möglichkeiten der Adressierung von Speicherstellen ist Thema von Kapitel 7.

Die beiden folgenden Kapitel behandeln ähnliche Mechanismen zur Strukturierung von Assembler-Programmen: Kapitel 8 beschäftigt sich mit Makros, in Kapitel 9 werden Unterprogramme und die Schnittstelle zum Betriebssystem MS-DOS behandelt.

Die Speichergrenze von 64 KByte wird in Kapitel 10 gesprengt, wo wir uns mit der Segmentierung von Assembler-Programmen beschäftigen.

Die Modularisierung von Assembler-Programmen, die in Kapitel 11 besprochen wird, ist Voraussetzung für das Kapitel 12, das die Schnittstelle zwischen Assembler und höheren Programmiersprachen behandelt. Bisher haben wir uns vorwiegend auf die Strukturierung des Programm-Codes konzentriert. In Kapitel 13 wird gezeigt, wie man im Assembler auch Daten strukturieren kann.

Alle Assembler-Programme, die wir bis hierher besprochen haben, können auf allen Rechner der Familie Intel 80x86 ausgeführt werden, da die Mitglieder dieser Familie aufwärts kompatibel sind, d.h. ein Nachfolge-Modell kennt alle Befehle seines Vorgängers und noch ein paar weitere Befehle. Diese zusätzliche Befehle werden wir im abschließenden Kapitel 14 behandeln.

In Anhang A sind die im Buch verwendeten Makros aufgelistet, die ebenfalls in der elektronisch verfügbaren Beispielsammlung enthalten sind.

Danksagung

Abschließend möchte ich meinem Kollegen Prof. Dr. Max Riederle herzlich danken, der das gesamte Manuskript der 1. Auflage kritisch durchgesehen und viele Verbesserungsvorschläge eingebracht hat. Viele Leserbriefe haben sich sehr positiv darüber geäußert, dass es eigentlich keine Druckfehler in diesem Buch gibt. Dies ist der akribischen Durchsicht des Manuskripts durch Max Riederle zu verdanken. Dafür nochmals – auch und insbesondere in der 5. Auflage – vielen Dank.

2 Die Entwicklungsumgebung

Bevor wir die Details der Assembler-Programmierung besprechen, soll hier an einem einfachen Problem die Entstehung eines Programms von der Idee bis zum ablauffähigen Code betrachtet werden. Dabei wird die Entwicklungsumgebung des Turbo Assemblers benutzt. Der Microsoft Assembler zusammen mit Codeview arbeitet entsprechend. Im einzelnen werden wir wie folgt vorgehen (siehe Abbildung 2.1):

- Nach der genauen Beschreibung des Problems, das wir lösen wollen, muss das Assembler-Programm entworfen werden. Da uns hierzu noch die Kenntnisse fehlen, liefert uns ein guter Geist ein fertiges Assembler-Programm. Dieses Programm muss nun mit Hilfe eines *Texteditors* eingegeben und in einer Datei abgespeichert werden.

- Wie im vorigen Kapitel erwähnt, bietet die Assemblersprache eine recht komfortable Möglichkeit, die Maschinenbefehle, die ja nur aus Nullen und Einsen bestehen, in einer für den Menschen lesbaren Form zu beschreiben. Damit das Assembler-Programm auch vom Rechner verstanden wird, muss es in die Maschinensprache übersetzt werden. Man sagt: „Das Assembler-Programm wird assembliert". Hierzu steht eine Anwendung zur Verfügung, die ein in Assembler geschriebenes Programm in eine so genannte Objekt-Datei übersetzt. Häufig nennt man dieses Übersetzungsprogramm auch Assembler, also ebenso wie die Sprache. Damit hier keine Verwechslungen auftreten, nennen wir im Folgenden das Übersetzungsprogramm *TMASM*, die Programmiersprache *Assembler*. TMASM soll andeuten, dass hier sowohl der Turbo Assembler wie auch der Microsoft Assembler verwendet wird.

- Die Objekt-Datei muss in ein ablauffähiges Programm überführt werden. Dazu müssen wir den *Linker* aufrufen. Dieser Linker erzeugt eine so genannte EXE-Datei (executable = ausführbar).

- Nun kann das Programm gestartet werden und löst hoffentlich unser Problem. Leider wird dieses Ziel in der Realität nur selten beim ersten Anlauf erreicht. Zum leichteren Auffinden von logischen Programmierfehlern können wir einen *Debugger* verwenden.

- Nach der Lokalisierung unserer Programmierfehler müssen wir unser Assembler-Programm verbessern, wozu wir erneut den Texteditor aufrufen. Der beschriebene Zyklus beginnt nun wieder von vorn.

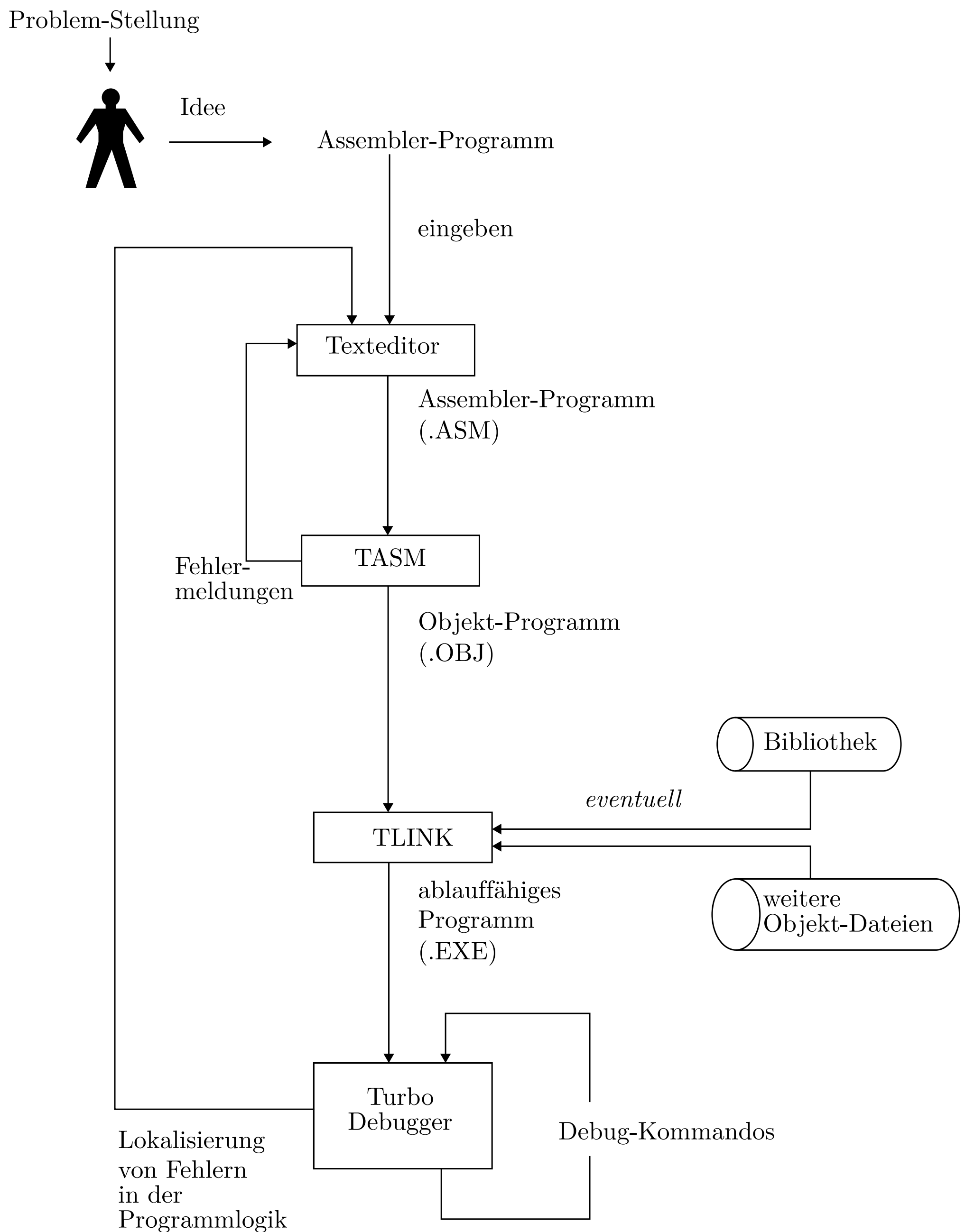

Abbildung 2.1: *Werdegang eines Programms*

2.1 Ein einfaches Beispiel

Unser erstes Assembler-Programm gibt auf dem Bildschirm einen Text aus:

```
**************************
*   Hallo, lieber Leser  *
**************************
```

Beim Studium dieses Buches werden wir lernen, welche Einzelschritte zum Erstellen dieses Programmes nötig sind. Im Moment soll ein fertiges Assembler-Programm vorgestellt werden, das mit einem Texteditor in den Rechner eingetippt und in eine Datei abgespeichert werden muss. Hierzu kann man jeden beliebigen Text-Editor verwenden. Das folgende Programm, das Sie jetzt eintippen sollten, löst unser Problem. Das Programm soll unter dem Namen GRUSS.ASM abgespeichert werden.

```
;*************************************
; unser erstes Assembler-Programm
;*************************************
        DOSSEG
        .MODEL  SMALL
        .STACK  100H
; jetzt kommen die Daten -------------
        .DATA
Hallo   db      10,13,10,13
        db      "**************************",10,13
Text    db      "*   Hallo, lieber Leser  *",10,13
        db      "**************************",10,13
Lnge    dw      3*28+4  ;Laenge des Textes

;jetzt kommt das Programm ------------
        .CODE
begin:  mov     ax,@Data
        mov     ds,ax
; Ausgabe des Textes 'Hallo' -----------
        mov     bx,1    ; Ausgabe auf Bildschirm
        mov     cx,Lnge ; Anz. auszugebende Zeichen
        mov     dx,Offset Hallo
; aus zugebende Zeichen-Reihe
        mov     ah,40h  ; Nr. der Ausgabe-Routine
        int     21h     ; Ausgabe starten

        mov     ax,4c00h
        int     21h     ; Programm beenden
        end     begin   ; Ende des Programm-Textes
```

Verglichen mit einem Pascal-Programm, das dieselbe Aufgabe löst, ist unser Assembler-Programm erschreckend lang. Das kommt einfach daher, dass wir jeden einzelnen Schritt beschreiben, den der Rechner ausführen muss. Bei der Eingabe haben wir im Editor

die Tabulatortaste benutzt, um das Programm formatiert einzugeben. Dies ist nicht zwingend vorgeschrieben, da der Assembler formatfrei ist. Für die Übersichtlichkeit ist dies aber mindestens ebenso wichtig wie die Kommentare, die hinter einem Semikolon stehen. Das Programm werden wir uns im nächsten Kapitel noch genauer anschauen.

2.2 Assembler und Linker

Nachdem das Programm in die Datei GRUSS.ASM gespeichert wurde, müssen wir es assemblieren. Wir verwenden im Folgenden die Werkzeuge, die mit dem Turbo Assembler ausgeliefert werden. Es wird folgendes Kommando benutzt:

```
TASM /zi GRUSS
```

Dieser Aufruf startet den Turbo Assembler mit der Eingabe GRUSS.ASM, der eine so genannte Objekt-Datei namens GRUSS.OBJ generiert. Diese Datei enthält im wesentlichen nur die Übersetzung der für uns leichter lesbaren Befehle in das zugehörige binäre Format. So wird etwa

```
mov     ah,40h
```

in die binäre Form

```
1011 0100 0100 0000
```

übersetzt. Bei obigem Kommando wurde die Option /zi mit angegeben. Diese Option ist notwendig, damit der Turbo Assembler zusätzlich Informationen in die Objekt-Datei aufnimmt, die zum Arbeiten mit dem Turbo Debugger nötig sind (siehe unten). Die Objekt-Datei ist noch nicht das ablauffähige Programm. Hierzu muss man erst noch den Binder (oder englisch Linker) starten, der aus einer oder mehreren Objekt-Dateien ein fertiges Programm erzeugt und in einer Datei mit der Erweiterung EXE ablegt. In unserem Beispiel erfolgt dies durch das Kommando

```
TLINK /v GRUSS
```

Auch hier dient die Option /v der Generierung von Zusatzinformation, die der Debugger benötigt. Jetzt müsste eine Datei GRUSS.EXE existieren, die wir von der Kommandozeile mit dem Befehl

```
GRUSS
```

einfach starten können. Dieser problemlose Werdegang eines Programms ist eine äußerst seltene Ausnahme. Im Normalfall erhalten wir schon beim Assemblieren diverse Warnungen und Fehlermeldungen. Diese Meldungen beschreiben in Englisch das entdeckte Problem zusammen mit der Angabe der Zeilennummer, auf die sich die Meldung bezieht. Die Fehlermeldungen müssen wir verstehen und die Fehler in unserer Assembler-Quelle verbessern. Fehler müssen korrigiert werden. Warnungen weisen auf kleinere Probleme hin, die TMASM entdeckt hat; das Programm muss deshalb nicht unbedingt falsch sein.

Meistens hat TMASM jedoch recht in seiner vagen Vermutung, dass das entdeckte Problem ein Programmierfehler ist. Jedenfalls sollte man jeder Warnung genau nachgehen. Nach der Ausmerzung der ersten Fehler wird erneut assembliert; dieser Zyklus wird dann solange wiederholt, bis keine Fehlermeldung mehr auftritt.

2.3 Der Debugger

Wenn unser Programm bis hierher ohne Fehlermeldung bearbeitet wurde, heißt das natürlich noch lange nicht, dass es auch tatsächlich das gestellte Problem löst. Man kann versuchen, sich durch systematische Testläufe von der Richtigkeit des Programms zu überzeugen.

Eine andere Möglichkeit, logische Fehler (engl. bugs) in einem Programm zu finden, bietet die Verwendung eines Debuggers. Zusammen mit dem Turbo Assembler wird der sehr komfortable Turbo Debugger geliefert. Er wird mit dem Kommando

```
TD GRUSS
```

gestartet. Ab der Version 2.0 unterstützt der Turbo Debugger die Maus; er kann aber auch sehr bequem über Tasten bedient werden. Der Turbo Debugger verfügt über eine kontext-abhängige Hilfe-Funktion, die größtenteils das Handbuch ersetzt. Deshalb wird hier auf eine ausführliche Besprechung dieses Programms verzichtet. Im Folgenden werden die beim Debuggen wichtigsten Begriffe erläutert.

Fenster-Technik

Nach dem Start des Turbo Debuggers erscheinen auf dem Bildschirm zwei Fenster: Das Modul-Fenster enthält den Quelltext unseres Assembler-Programms, im unteren Watch-Fenster kann man den Wert von Variablen beobachten. Bei den meisten Befehlen öffnet der Turbo Debugger ein weiteres Fenster, in dem Informationen angezeigt werden oder ein Dialog mit dem Benutzer geführt wird. Man wechselt in ein anderes Fenster, indem man es mit der Maus anklickt oder zusammen mit der Alt-Taste die Ziffer eingibt, die in der rechten oberen Ecke des Fensters steht.

Wenn das ablaufende Programm nun selbst Ausgaben auf den Bildschirm bringen will, könnte es mit dem Debugger, der ja schon den gesamten Bildschirm beansprucht, ins Gehege kommen. Dieses Problem wird vom Turbo Debugger so gelöst, dass er einen virtuellen Bildschirm verwaltet, den man sich nach Drücken der Tasten Alt+F5 anschauen kann. Mit einem weiteren beliebigen Tastendruck kommt man wieder zum Debugger zurück.

Variablen-Werte

In dem standardmäßig geöffneten Watch-Fenster kann man den Wert von Variablen ständig beobachten. Dazu muss man lediglich ins Watch-Fenster springen und den gewünschten Variablen-Namen eintippen.

Kompliziertere Datenstrukturen kann man über den Menü-Punkt *Data-Inspect* bis ins letzte Detail anschauen. Sollte man dabei feststellen, dass der Wert einer Variablen nicht stimmt, kann er über den Menü-Punkt *Data-Evaluate/modify* sogar verändert werden.

Haltepunkte

Eine der wesentlichen Aufgaben eines Debuggers ist es, die Abarbeitung eines Programms an einer bestimmten Stelle zu unterbrechen und dem Benutzer die Möglichkeit zum Eingreifen zu geben. Solche Unterbrechungsstellen heißen *Haltepunkt* oder *Breakpoint.* Der Turbo Debugger kennt zwei unterschiedliche Typen von Haltepunkten:

- *Unbedingte Haltepunkte* sind Programm-Stellen, an denen die Ausführung immer unterbrochen werden soll. Man kann hierbei noch festlegen, dass das Programm erst unterbrochen wird, nachdem die angegebene Programm-Stelle eine bestimmte Anzahl mal durchlaufen worden ist. Im Breakpoint-Menü werden unbedingte Haltepunkte über *Toggle* bzw. *At...* definiert.
- Ein *bedingter Haltepunkt* wird mit einer Bedingung gekoppelt. Das Programm wird nur dann angehalten, wenn die angegebene Bedingung zutrifft. Eine Standard-Aufgabe beim Debuggen ist das Auffinden der Programm-Stelle, an der eine bestimmte Variable unbeabsichtigt überschrieben wird. In diesem Fall gibt man den Variablen-Namen über den Menü-Punkt *Breakpoints-Changed memory global* ein. Will man wissen, an welcher Stelle eine Variable einen bestimmten Wert annimmt, kann man dies im Breakpoints-Menü unter dem Punkt *Expression true global* eingeben.

Einzelschritt

Wenn man noch überhaupt keine Ahnung hat, warum das Programm Mist macht, wird man sich die Abarbeitung Schritt für Schritt anschauen wollen. Mit der Taste F7 wird jeweils die nächste Programm-Zeile abgearbeitet – das ist die Zeile, die ganz links mit einem Dreieck markiert ist.

Will man sich das dauernde Drücken der Taste F7 ersparen, kann man sich die Abarbeitung des Programms wie einen Film Befehl für Befehl vorspielen lassen. Hierzu muss man nur im Menü *Run* den Punkt *Animate* anwählen und die gewünschte Geschwindigkeit einstellen. Ein beliebiger Tastendruck unterbricht die Film-Vorführung.

Das CPU-Fenster

In den nächsten Kapiteln werden wir uns mit den prozessor-nahen Komponenten wie den Registern und dem Keller beschäftigen. Diese Komponenten sind für einen Pascal- oder C-Programmierer meistens uninteressant. Deshalb zeigt der Turbo Debugger diese Interna des Rechners auch nur auf Wunsch, nämlich im CPU-Fenster. Dieses Fenster wird im Menü-Punkt *View-CPU* geöffnet. Es erscheint ein fünf-geteiltes Fenster:

- Im größten Teil des Fensters links oben erscheint wieder der Quelltext, diesmal aber versehen mit seltsamen Zahlen. Diese Zahlen geben die Positionen der Befehle

im Speicher sowie ihre interne Verschlüsselung wieder. Der Aufbau entspricht der Form des Assembler-Listings, das wir in Abschnitt 3.7 besprechen werden.

- Im rechten oberen Teil sind die Register mit ihren Inhalten aufgeführt. Über Register werden wir in Kapitel 5 noch sprechen.
- Der interne Zustand des Rechners wird durch so genannte Flags beschrieben, deren Werte wir ganz rechts oben sehen.
- Im linken unteren Teil-Fenster kann man sich einen Ausschnitt des Speicher-Inhalts anschauen.
- Der rechte untere Teil zeigt das obere Ende des Kellers; das oberste Keller-Element ist mit einem Dreieck markiert. Der Keller wird intensiv bei Unterprogrammen benutzt, die wir in Kapitel 9 besprechen werden.

Mit der Tabulator-Taste oder der Maus kann man von einem Teilfenster zum anderen springen. Die Inhalte der Register kann man auch verändern. Dazu muss man das gewünschte Register mit der Maus oder den Pfeiltasten anwählen und den neuen Wert einfach eintippen.

Mit der Tasten-Kombination Alt-X wird der Turbo Debugger verlassen.

3 Allgemeiner Aufbau eines Assembler-Programms

Wie wir in unserem Beispiel aus dem vorigen Kapitel gesehen haben, sind Assembler-Programme zeilen-orientiert, d.h. die elementaren Einheiten sind Zeilen. Anders als in den meisten höheren Programmiersprachen, in denen in einer Zeile mehrere Anweisungen stehen können oder auch eine Anweisung sich über mehrere Zeilen erstrecken kann, enthält eine Zeile eines Assembler-Programms stets eine komplette Anweisung. Eine besonders lange Anweisung kann man auch über mehrere Zeilen schreiben: Dazu muss man am Ende der Zeile das Zeichen `\` angeben, wenn die Anweisung auf der nächsten Zeile fortgesetzt werden soll. Übersetzt TMASM das Programm, so bearbeitet er eine Anweisung nach der anderen. Die Wirkung dieser Bearbeitung kann man in zwei Klassen unterteilen:

- Die erste Klasse von Assemblerzeilen erzeugt aus den Abkürzungen der Maschinenbefehle das entsprechende Bitmuster, das dann später, wenn das fertige Programm gestartet wird, vom Prozessor ausgeführt wird. Solche Assemblerzeilen wollen wir *Befehlszeilen* nennen.
- Die zweite Klasse von Assemblerzeilen steuert die Aktivitäten von TMASM, d.h. sie haben nur während der Übersetzungszeit eine Wirkung. Solche Assemblerzeilen werden wir *Direktivenzeilen* nennen.

Mit diesen beiden Begriffen können wir festhalten:

> Ein Assembler-Programm ist eine Folge von Befehls- und Direktivenzeilen.

Diese Definition ist sicherlich noch nicht komplett. Für den Moment reicht sie uns aber, um den formalen Aufbau eines Assembler-Programms zu besprechen.

Obwohl der syntaktische Aufbau von Befehlen und Direktiven sehr ähnlich ist, wollen wir sie getrennt behandeln, um die Unterschiede klarer herausarbeiten zu können.

3.1 Bezeichner

Zur Benennung eines Speicherplatzes kann ein Bezeichner verwendet werden, der wie folgt aufgebaut ist:

Bezeichner:
besteht aus einer Folge von • Buchstaben `a` bis `z` (ohne Umlaute und ß), • Buchstaben `A` bis `Z` (ohne Umlaute), • Zeichen `@`, `$`, `_`, `?`, • am Anfang eines Bezeichners darf ein Punkt vorkommen, • Ziffern `0` bis `9`.
Einschränkungen: • Erstes Zeichen darf keine Ziffer sein. • `$` bzw. `?` ohne ein weiteres Zeichen ist nicht erlaubt. • Ein Bezeichner darf kein reserviertes Wort sein. • Zwischen Groß- und Kleinschreibung wird nicht unterschieden.
Bemerkungen: • Bezeichner können beliebig lang sein; Turbo Assembler unterscheidet sie alle. (Dagegen unterscheidet der Microsoft Assembler MASM nur maximal 31 Zeichen.) • In einem Bezeichner darf kein Leerzeichen oder Zeilenwechsel vorkommen.

□ **Beispiel 3.1.1**

Korrekte Bezeichner sind:

```
A
Wort
?Alles_klar?
@NaSowas
Erste_Variable
ERSTE_VARIABLE
```

Die letzten beiden Bezeichner sind für den Assembler identisch.

Keine Bezeichner sind:

```
1.Variable
'Name'
Lügenmärchen
MOV (reserviertes Wort)
```

Bezeichner kommen im Assembler außer zur Benennung von Speicherplätzen noch an vielen anderen Stellen vor, die wir noch kennenlernen werden.

Im weiteren Verlauf des Buches werden wir viele neue Begriffe einführen, wobei wir ein einheitliches Schema verwenden. Das Schema hat folgende Form:

neuer Begriff:
definierende Regel
Wirkung: • Beschreibung der Wirkung des neuen Begriffs.
Bemerkung(en): • weitere Bemerkungen zum neuen Begriff.

Die Rubrik Wirkung erklärt, was der neue Begriff bewirkt. Die Rubrik Bemerkung(en) enthält Erläuterungen zu dem neuen Begriff. Jede dieser beiden Rubriken kann auch fehlen.

Für die definierenden Regeln wird folgende Notation festgelegt:

- *Terminale Symbole* sind solche, die in der angegebenen Form direkt im Assembler-Programm vorkommen; sie werden in der Schrift `Courier` geschrieben.
- *Nichtterminale Symbole* sind solche, die durch zusätzliche Schemata oder erklärenden Text weiter definiert werden.
- Alternativen werden durch den senkrechten Strich | getrennt.
- Optionale Teile werden in kursiven eckigen Klammern *[. . .]* eingeschlossen.
- Zur besseren Lesbarkeit können Teile mit kursiven runden Klammern *(. . .)* geklammert werden.
- Die Schreibweise $(\ldots)^*$ bedeutet, dass der geklammerte Teil keinmal, einmal oder beliebig oft vorkommt.
- Die Schreibweise $(\ldots)^+$ bedeutet, dass der geklammerte Teil mindestens einmal oder beliebig oft vorkommt.

3.2 Befehle

Eine Befehlszeile ist wie folgt aufgebaut:

Befehlszeile:
`[Marke:] [Befehl [Operanden]] [;Kommentar]`

Jede einzelne Komponente ist in eckige Klammern gesetzt, um auszudrücken, dass sie auch fehlen kann. Fehlen alle Komponenten, haben wir eine *Leerzeile*; fehlt alles außer dem Kommentar, so haben wir eine *Kommentarzeile*.

Zwischen den einzelnen Komponenten stehen einige Leerzeichen. Der Assembler verlangt hier lediglich, dass zwischen zwei vorkommenden Komponenten mindestens ein Leerzeichen oder Tabulatorzeichen stehen muss. Es empfiehlt sich jedoch aus Gründen der Lesbarkeit, für die einzelnen Komponenten einer Befehlszeile möglichst feste Spalten einzuhalten. Eine Kommentarzeile darf und wird häufig mit einem Semikolon auf Spalte 1 beginnen.

Nun zu den Komponenten der Befehlszeile:

- Eine `Marke` ist ein Bezeichner, gefolgt von einem Doppelpunkt. Sie kennzeichnet eine Speicherstelle im Befehlsteil des Programms. Marken werden meist als Sprungziele verwendet. Folglich darf es keine zwei Marken-Definitionen mit demselben Bezeichner geben. Eine Marke darf auch alleine auf einer Zeile stehen. In diesem Fall bezieht sich die Marke dann auf die nächste Befehlszeile.
- `Befehle` sind lesbare und leicht verständliche Kurzbezeichnungen für Befehle der Maschinensprache; sie gehören zu den reservierten Wörtern des Assemblers. Bei der Besprechung unseres Beispiels werden wir einige Befehle kennenlernen.
- Während der Befehl festlegt, *was* der Rechner tun soll, geben die `Operanden` an, *womit* dies getan werden soll. Operanden können u.a. Konstanten, Bezeichner für Variablen oder Register sein. Die Anzahl der Operanden hängt von der Art des Befehls ab; sie kann zwischen keinem und zwei variieren. Benötigt ein Befehl zwei Operanden, werden diese durch ein Komma getrennt. TMASM übersetzt den Befehl zusammen mit den notwendigen Operanden in einen Maschinenbefehl, dessen interne Bytelänge wieder vom Typ des Befehls abhängt.
- Hinter dem Befehl und seinen Operanden kann ein `Kommentar` folgen, der durch ein Semikolon eingeleitet wird und bis zum Zeilenende reicht. Der Kommentar wird von TMASM überlesen.

☐ **Beispiel 3.2.1**

```
Bsp1:   mov ah,Bu    ; weist ah Inhalt von Bu zu
        add ah,3     ; erhoeht ah um 3
```

3.3 Direktiven

Eine Direktivenzeile hat einen sehr ähnlichen formalen Aufbau wie eine Befehlszeile:

Direktivenzeile:
`[[Name]     Direktive     [Operanden]]        [;Kommentar]`

Auch hier sind wieder alle Komponenten optional; für die Trennung der einzelnen Komponenten gilt dieselbe Regel wie bei den Befehlszeilen. Fehlen alle Komponenten bis auf den Kommentar, so haben wir eine Kommentarzeile; fehlen alle Komponenten, so liegt eine Leerzeile vor. Die einzelnen Komponenten haben folgende Bedeutung:

- Ein `Name` ist ein Bezeichner, dem hier *kein* Doppelpunkt folgen darf. Je nach Direktive kennzeichnen solche Namen ebenfalls Speicherstellen, z.B. von Variablen, manchmal haben sie auch eine andere Bedeutung.
- Der Assembler kennt eine Fülle von `Direktiven`, die den Assemblierungsprozeß steuern. Die Direktiven gehören ebenfalls zu den reservierten Wörtern des Assemblers und sind im Register unter dem Stichwort Direktive zusammengestellt.
- Die Anzahl der auftretenden `Operanden` variiert hier viel stärker als in Befehlszeilen. Die einzelnen Operanden werden durch Kommata getrennt.
- Auch Direktivenzeilen können durch einen `Kommentar` abgeschlossen werden, der wieder durch ein Semikolon eingeleitet wird.

Direktiven werden wir noch häufig im Verlauf dieses Buches antreffen. Um eine ungefähre Vorstellung von der Einsatzbreite der Direktiven zu erhalten, wollen wir hier nur die Hauptaufgaben von Direktiven summarisch aufzählen und auf die Stellen verweisen, in denen diese Direktiven behandelt werden:

- Speicher-Reservierung und Konstanten-Definitionen (Kapitel 4).
- Organisation des Speichers für das gesamte Programm (Kapitel 10).
- Steuerung der Generierung einer Listing-Datei (Abschnitt 3.7).
- Steuerung der bedingten Assemblierung (Kapitel 8).
- Makro-Definitionen und -Aufrufe (Kapitel 8 und Abschnitt 3.6).

☐ **Beispiel 3.3.1**

```
Buchst  db  'A'     ; Speicherplatz fuer Buchstaben
        db  "BCD"   ; weiterer Speicherplatz
```

Während in der ersten `db`-Direktive genau ein Byte reserviert wurde, belegt die zweite drei Bytes, nämlich für jeden Buchstaben der angegebenen Zeichenreihe ein Byte. ■

3.4 Ein Standard-Rahmen für Assembler-Programme

Fast alle unsere Beispiel-Programme haben einen festen Rahmen von Direktiven, die das Assembler-Programm klammern und die Übersetzung steuern. Wie in unserem Beispiel aus Kapitel 2 hat dieser Rahmen folgende Form:

```
      DOSSEG
      .MODEL  SMALL
      .STACK  100H
; jetzt kommen die Daten --------------------------
      .DATA
; Datenvereinbarungen ...

; jetzt kommt das Programm------------------------
      .CODE
begin:

 ; hier stehen die Befehle ...

      end     begin   ; Ende des Programmtextes
```

Zur Erläuterung des Programm-Rahmens müssen wir etwas ausholen und auf die besondere Situation von 16 Bit-Rechnern eingehen. Die Speicherstellen unseres Rechners werden zum Abspeichern von Befehlen, Ergebnissen und Zwischen-Ergebnissen benutzt. Um auf den Speicher gezielt zugreifen zu können, muss jede einzelne Speicherstelle identifizierbar sein. Intern erhält jede Gruppe von 8 Bit (= 1 Byte) im Speicher ihre eigene *Hausnummer* oder *Adresse*. Wenn wir nun eine 16 Bit-Zahl für die Adressierung zur Verfügung haben, so stellen wir fest, dass wir damit genau $2^{16} = 65536$ verschiedene Bytes adressieren können. Dies ist eine Speichermenge von 64 KByte (Kilo-Byte). Nun hat aber unser Rechner, je nach Ausbau, zwischen 256 KByte und 640 KByte oder sogar noch mehr Speicher. Wie kann man auf die restlichen Speicherstellen zugreifen?

Eine Möglichkeit besteht darin, die Speicherung des Programms so aufzuteilen, dass die Befehle in einem ersten Speicherbereich, die Daten in einem zweiten und die Arbeitsdaten in einem dritten Speicherbereich untergebracht werden. Der Speicherzugriff kann dann über die 16 Bit-Adresse und mit dem Zusatz Befehl, Daten bzw. Arbeitsdaten erfolgen. Solche Speicherbereiche nennt man im Assembler *Segmente*.

Diese drei Segmente werden dann beim Übersetzen in einer bestimmten Reihenfolge in *einer* Datei, der EXE-Datei, abgespeichert. Da wir beim Assembler-Programmieren jedes Detail beschreiben können, haben wir auch auf die Festlegung dieser Reihenfolge Einfluss.

Nun zurück zu unserem Beispiel. Fast immer ist eine Reihenfolge ebenso gut wie eine andere, Hauptsache, das Programm kennt sich aus. Deshalb wurde eine Standard-Reihenfolge für die drei Segmente festgelegt, die wir in unserem Programm durch die Direktive

```
    DOSSEG
```

ausgewählt haben. Die genaue Art der Adressierung legt die Direktive

```
.MODEL   SMALL
```

fest. Andere so genannte *Speichermodelle* werden wir in Kapitel 10 behandeln.

Um dem Assembler mitzuteilen, welche Assemblerzeilen in welchen Speicherbereich gehören, haben wir drei weitere Direktiven:

```
.DATA
```

zeigt an, dass die folgenden Zeilen in das Daten-Segment gehören. Die Direktive

```
.CODE
```

legt entsprechend fest, dass nun Anweisungen folgen, die ins Code-Segment gehören. Die Berechnung und Verwaltung von Arbeitsdaten werden von TMASM selbständig erledigt; wir müssen im Programm lediglich festlegen, wie viel Speicherplatz hierfür vorgesehen wird. Dafür sorgt die Direktive

```
.STACK  100h
```

die 100h Bytes = 256 Bytes hierfür reserviert. Diese Größe ist für praktisch jedes mittelgroße Assembler-Programm ausreichend.

In einem Programm können mehrere Direktiven `.CODE` und `.DATA` vorkommen; damit kann man zwischen den beiden Speicherbereichen umschalten. Andererseits darf es nur *eine* Direktive `.STACK` in einem Programm geben.

Jedes Assembler-Programm muss genau eine `END`-Direktive enthalten, die das Ende des Quelltextes kennzeichnet. Alle Zeilen, die hinter der `END`-Direktive stehen, werden vom Assembler ignoriert. Zusätzlich legt diese Direktive die *Startadresse* fest: Das ist in unseren Beispielen die Marke `begin`, die die Stelle bezeichnet, an der das Programm zur Ausführungszeit gestartet wird. Häufig ist dies die erste Anweisung hinter der Direktive `.CODE`. Fehlt bei der `END`-Direktive eine Marke, wird bei der ersten Anweisung des Assembler-Programms begonnen.

3.5 Befehle und Direktiven des Beispielprogramms

Im Folgenden soll eine informelle, aber ausführliche Erklärung unseres Beispielprogramms aus Kapitel 2 gegeben werden. Die Details werden wir in späteren Kapiteln genauer besprechen. Unser Beispiel enthält im Datenteil folgende Zeilen:

```
Hallo   db      10,13,10,13
        db      "**************************",10,13
Text    db      "*   Hallo, lieber Leser  *",10,13
        db      "**************************",10,13
Lnge    dw      3*28+4  ;Laenge des Textes
```

Dadurch werden drei Variable eingeführt: `Hallo`, `Text` und `Lnge`. Die Direktiven hinter den Variablen geben den Datentyp an: So können die Variablen `Hallo` und `Text` Zeichenreihen bzw. Folgen von Bytes aufnehmen, was durch die Direktive

```
db   (define Byte)
```

angegeben wird. Die Variable `Lnge` kann ein oder mehrere Daten des Typs Wort (= 16 Bit) aufnehmen, was durch die Direktive

```
dw   (define Word)
```

festgelegt wird.

Die Variablen sind hier alle initialisiert. Beginnen wir mit der Variablen `Text`: Sie erhält den Wert

```
*   Hallo, lieber Leser  *
```

gefolgt von zwei Buchstaben mit den Werten 10 und 13. Diese beiden Werte sind im ASCII-Code, mit dem der Assembler arbeitet, die Zeichen Zeilenwechsel(Line feed = 10) und Wagenrücklauf (carriage return = 13). Die nächste Zeile mit den vielen Sternen gehört auch noch zu der Variablen `Text`.

Entsprechend ist der Inhalt von `Hallo` zweimal das Paar Zeilenwechsel, Wagenrücklauf, gefolgt von Sternen und nochmals Zeilenwechsel, Wagenrücklauf. Dabei taucht ein Problem auf: Wie viel Speicherplatz belegt nun eigentlich eine Variable? Anders als in höheren Programmiersprachen bezeichnen die obigen Variablen-Namen lediglich die Speicherstelle, an der ihr Inhalt *beginnt.* Wie lang die Zeichenreihe ist, wird extra angegeben. Hierzu verwenden wir die Variable `Lnge`, die mit der ganzen Zahl `3*28+4 (=88)` vorbelegt ist.

Bevor wir den Befehlsteil genauer diskutieren, müssen wir uns kurz die Arbeitsweise unseres Prozessors Intel 8086 anschauen. Er besitzt einige *Register*. Das sind spezielle Speicherzellen, auf die sehr schnell zugegriffen werden kann. Im Moment reicht es zu wissen, dass es die 16-Bit Register `ax, bx, cx, dx` und `ds` gibt, wobei das letzte zur Adressierung der Daten benutzt wird. In unserem Beispiel kommt noch das Register `ah` vor; das ist nichts anderes als die linke Hälfte des Registers `ax`, die ein Byte aufnimmt. Eine vollständige Behandlung der Register folgt in Kapitel 5.

Nun zurück zu unserem Beispiel. Die ersten beiden Befehle

```
begin:  mov     ax,@Data
        mov     ds,ax
```

enthalten beide den Befehl `mov`, dessen Wirkungsweise wir deshalb zunächst allgemein besprechen wollen. Der Befehl

```
mov     ziel,quelle
```

überträgt den Inhalt von `quelle` nach `ziel`. Dabei können `quelle` und `ziel` jeweils ein Register oder ein Name sein, der eine Speicherzelle benennt. Zusätzlich kann `quelle` auch direkt eine Konstante sein. Der Befehl `mov` realisiert also eine *Wertzuweisung*.

Damit können wir die beiden Befehle besprechen: Der erste Befehl weist dem Register `ax` den Wert `@Data` zu; dies ist ein vom Assembler vordefinierter Bezeichner, der auf den *Anfang des Datenbereichs* zeigt. Im zweiten Befehl wird nun der Wert von `ax`, nämlich `@Data`, in das Segment-Register `ds` geladen.

Diese Befehlssequenz sieht etwas umständlich aus; man könnte doch gleich den Wert von `@Data` an das Segment-Register zuweisen. Das lässt der Assembler aber nicht zu, da der `mov`-Befehl ein paar Einschränkungen hat, die wir in Kapitel 5 genau besprechen werden.

Die beiden oben angegebenen Befehle *müssen* am Anfang eines jeden Programmes stehen, wenn im Programm Daten vorkommen. Andernfalls können wir nämlich auf keine Daten im Datenbereich zugreifen.

```
; Ausgabe des Textes 'Hallo' ----------------------
      mov     bx,1    ; Ausgabe auf Bildschirm
      mov     cx,Lnge ; Anz. auszugebende Zeichen
      mov     dx,Offset Hallo
                      ; auszugebende Zeichenreihe
      mov     ah,40h  ; Nr. der Ausgabe-Routine
      int     21h     ; Ausgabe starten
```

Das Schlüsselwort `Offset` gibt an, dass wir in dieser Befehlsfolge die Adresse der Variablen `Hallo` verwenden wollen. Der letzte Befehl hilft uns, ein Problem zu lösen: Wie funktioniert der Zugriff auf den Bildschirm oder ein anderes Peripherie-Gerät? Dieses Problem scheint, wenn wir an höhere Programmiersprachen wie z.B. Pascal denken, recht trivial zu sein. In Pascal gibt es die Prozeduren `write` und `writeln`, die die als Argument angegebene Zeichenreihe einfach auf den Bildschirm ausgeben.

Aber erinnern wir uns: Beim Assembler-Programmieren befinden wir uns ganz dicht am Prozessor; jede seiner Aktivitäten muss programmiert werden. Wie weiß nun der Prozessor, ob und wo sich ein spezielles Peripheriegerät befindet und wie es anzusprechen ist? Bei der Beantwortung dieser Frage hilft uns das Betriebssystem MS-DOS, unter dem TMASM selbst läuft. MS-DOS hat nämlich dieselben Probleme wie wir und diese bereits durch spezielle Unterprogramme, die so genannten DOS-Funktionen, gelöst. Diese Funktionen können wir mit

```
    int 21h
```

aufrufen. Die einzelnen DOS-Funktionen sind durchnumeriert; die Nummer der gewünschten Funktion ist stets vor dem Aufruf in das Register `ah` zu laden. In unserem Beispiel wollen wir eine Zeichenreihe ausgeben, was die DOS-Funktion `40h = 64` erledigt. Je nach Funktion werden weitere Angaben benötigt. In unserem Beispiel etwa:

- *Wohin soll ausgegeben werden?* Eine Codierung des Gerätes steht in `bx`, hier `1` für den Bildschirm. Lädt man nach `bx` den Wert `4`, so wird der Drucker als Ausgabe-Gerät gewählt.
- *Was soll ausgegeben werden?* Hier geben wir hinter dem Schlüsselwort `Offset` den Namen der Variablen `Hallo` an, an der der auszugebende Text beginnt. Die DOS-Funktion erwartet die Adresse im Register `dx`.

- *Wie viel soll ausgegeben werden?* Diese Information erwartet die DOS-Funktion im Register `cx`, die wir mit dem Wert der Variablen `Lnge` laden.

Damit ist unser Problem bereits gelöst, und wir müssen nur noch unser Programm sauber beenden, so dass sich anschließend wieder das Betriebssystem meldet. Dies wird mit der DOS-Funktion `4Ch` erledigt, die in der letzten Befehlsfolge aufgerufen wird. Man beachte, dass `ah` die linke Hälfte von `ax` ist, der `mov`-Befehl also die richtige Funktionsnummer lädt.

```
mov ax,4c00h
int 21h ; Programm beenden
end begin ; Ende des Programmtextes
```

Mit dieser Befehlsfolge muss jedes Assembler-Programm beendet werden. In unserem Beispiel haben wir zwei DOS-Funktionen verwendet. In Abschnitt 9.7 werden einige der wichtigsten DOS-Funktionen genauer behandelt.

3.6 Makro-Aufrufe

In unserem ersten Programm haben wir gesehen, dass man im Assembler jede Kleinigkeit ganz detailliert beschreiben muss. Um unsere weiteren Beispiele einigermaßen übersichtlich zu halten, haben wir für häufig wiederkehrende Standard-Aufgaben so genannte *Makros* definiert. Ein Makro sieht wie ein einzelner Befehl aus. Hinter ihm verbirgt sich aber eine mehr oder weniger lange Folge von Befehlen.

Im Anhang A sind die vollständigen Definitionen der Makros aufgelistet. In den herunterladbaren Programmbeispielen (siehe letzte Seite) sind sie in der Datei `macros.mac` zu finden. Nach dem Studium des Makro-Konzepts, das in Kapitel 8 besprochen wird, werden Sie diese Makro-Definitionen vollständig verstehen.

Im Folgenden wird beschrieben, wie die Makros verwendet werden und was sie leisten. An kurzen Beispielen – meist Erweiterungen unseres Beispiels aus Kapitel 2 – wird ihr Einsatz verdeutlicht.

Bei der Beschreibung der Makros sind die durch eckige Klammern `[ ]` eingeklammerten Parameter optional, d.h. diese Parameter können beim Aufruf angegeben werden oder nicht. Für jeden fehlenden optionalen Parameter ist ein bestimmter Wert vordefiniert.

3.6.1 Ausgabe von Text

Im Beispiel aus Abschnitt 2.1 wurde ein Text auf den Bildschirm ausgegeben. Dabei war, wie wir im vorigen Abschnitt besprochen haben, die Definition des Textes im Datenteil angegeben, die Befehle für die Ausgabe stehen im Programmteil. Diese beiden Stellen können in einem größeren Programm weit auseinander liegen. Es wäre sowohl bequemer wie auch der Lesbarkeit des Programmes förderlich, wenn der auszugebende Text und die Befehle für die Ausgabe an derselben Stelle angegeben werden könnten. Dies leistet das Makro `print`.

`print`	Ausgabe eines Textes ohne Zeilenwechsel
`println`	Ausgabe eines Textes mit Zeilenwechsel
`nl`	Ausgabe eines Zeilenwechsels

Aufruf:

`print`	*Text[, Ausgabe]*
`println`	*[Text][, Ausgabe]*
`nl`	

Wirkung:

- Der *Text*, der in Apostrophs oder Anführungszeichen eingeschlossen sein muss, wird ausgegeben.
- Der Parameter *Ausgabe* bestimmt, wohin der Text ausgegeben wird. Es sind nur die beiden folgenden Werte erlaubt:

  1 : Bildschirm (Voreinstellung)

  4 : Drucker
- Wird `println` ohne den Parameter *Text* verwendet, wird nur ein Zeilenwechsel ausgegeben.
- `nl` gibt einen Zeilenwechsel auf dem Bildschirm aus.

Alle in diesem Abschnitt besprochenen Makros sind in der Datei `macros.mac` abgespeichert. Damit unsere Assembler-Programme die dort definierten Makros verwenden können, muss zu Beginn die Zeile

```
include macros.mac
```

angegeben werden.

Unter Verwendung der Makros `print` und `println` kann unser Beispiel aus Abschnitt 2.1 wie folgt umgeschrieben werden:

□ Beispiel 3.6.1

```
        DOSSEG
        .MODEL  SMALL
        .STACK  100H
Ausg    equ     1   ; fuer Bildschirm
         ; oder 4     fuer Drucker
include macros.mac
        .CODE
begin:  mov     ax,@Data
        mov     ds,ax

        println ,Ausg
        println "*************************",Ausg
```

```
        println "*   Hallo, lieber Leser  *",Ausg
        println "************************",Ausg

        mov     ax,4c00h
        int     21h
        end     begin
```

■

Ist die Zeichenreihe wie im Anfangsbeispiel definiert, haben wir ein Problem: Wir müssen der DOS-Funktion für die Ausgabe neben der Anfangsadresse der auszugebenden Zeichenreihe auch deren Länge übergeben. Es wäre bequem, wenn diese Länge automatisch berechnet würde. Aber irgendwie muss man die Länge oder das Ende der Zeichenreihe kennzeichnen. Schielen wir zu den Compilern für Pascal und C++ hinüber, sehen wir, dass jeder Compiler hierfür ein eigenes Verfahren benutzt. Wir wollen bei unseren Makros die Konvention von C++ übernehmen:

Eine Zeichenreihe wird mit einem NUL-Byte (ASCII-Zeichen 0) abgeschlossen.

Das folgende Makro gibt solche NUL-terminierten Zeichenreihen aus.

`writeS`	Ausgabe einer Zeichenreihe
Aufruf: `writeS`	*TextVar[, Ausgabe]*

Wirkung:

- Der ab der Adresse *TextVar* stehende Text wird bis zum ersten NUL-Byte ausgegeben.
- Der Parameter *Ausgabe* bestimmt, wohin der Text ausgegeben wird. Es sind nur die beiden folgenden Werte erlaubt:

  1 : Bildschirm (Voreinstellung)
  4 : Drucker

□ Beispiel 3.6.2

```
        DOSSEG
        .MODEL  SMALL
        .STACK  100H
include macros.mac
        .DATA
Hallo   db      10,13,10,13
        db      "************************",10,13
Text    db      "*   Hallo, lieber Leser  *",10,13
        db      "************************",10,13
```

```
        db      0       ; <<-- Text-Ende
        .CODE
begin:  mov     ax,@Data
        mov     ds,ax

        writeS  Hallo

        mov     ax,4c00h
        int     21h     ; Programm beenden
        end     begin   ; Ende des Programmtextes
```
■

3.6.2 Einlesen von Text

Will man eine Zeichenreihe über die Tastatur einlesen, braucht man einen Speicherbereich, in den man die eingelesene Zeichenreihe abspeichern kann. Im folgenden Beispiel soll in den Speicherbereich, dessen Anfang mit Text gekennzeichnet ist und der mit

```
*   Hallo, lieber Leser  *
```

initialisiert ist, etwas eingelesen werden. Dazu verwenden wir das folgende Makro.

`readS`	Zeichenreihe einlesen
Aufruf: `readS`	*TextVar[, [ELnge] [,ALnge]]*

Wirkung:

- Es werden Zeichen von der Tastatur eingelesen und in den Speicherbereich ab *TextVar* abgelegt. Die Tastatur-Eingabe wird mit der Taste RETURN abgeschlossen.
- Der optionale Parameter *ELnge* gibt an, wie viele Zeichen maximal eingelesen werden dürfen. Werden mehr Zeichen eingegeben, werden die überzähligen ignoriert. Voreinstellung für *ELnge* ist 255.
- Der optionale Parameter *ALnge* muss eine 16 Bit-Variable sein, die nach der Eingabe die Anzahl der tatsächlich eingegebenen Zeichen aufnimmt.

Bemerkung:

- Fehlt der Parameter *ELnge*, soll aber *ALnge* angegeben werden, müssen zwei Kommata geschrieben werden, z.B.: `readS name , , Lge`

□ **Beispiel 3.6.3**

```
        DOSSEG
        .MODEL  SMALL
```

```
        .STACK  100H
include macros.mac
        .DATA
Hallo   db      10,13,10,13
        db      "************************",10,13
Text    db      "*   Hallo, lieber Leser  *",10,13
        db      "************************",10,13
        db      0       ; <<-- Text-Ende
Lnge    dw      ?
        .CODE
begin:  mov     ax,@Data
        mov     ds,ax

        print   "Bitte maximal 31 Zeichen eingeben : "
        readS   Text,31,Lnge
        writeS  Hallo
        mov     ax,4c00h
        int     21h     ; Programm beenden
        end     begin   ; Ende des Programmtextes
```

Werden mehr als 31 Zeichen eingelesen, werden die überzähligen Zeichen einfach abgeschnitten. Versuchen Sie einmal, beim Makro `readS` nur den ersten Parameter anzugeben und mehr als 31 Zeichen einzugeben. Aber geben Sie nicht zu viele Zeichen ein, denn Ihr Programm könnte sonst abstürzen! ■

3.6.3 Einlesen und Ausgeben von Zahlen

Im vorigen Beispiel wurde in der Variablen `Lnge` die Anzahl der tatsächlich eingegebenen Zeichen abgespeichert. Diesen Zahlenwert wollen wir auf dem Bildschirm ausgeben. Er ist in einem 16 Bit-Wort als Dualzahl abgespeichert. Auf dem Bildschirm können aber nur ASCII-Zeichen dargestellt werden. Um das Problem zu verdeutlichen, betrachten wir ein Beispiel: Wurden 13 Zeichen eingegeben, so enthält die Variable `Lnge` die Dualzahl

```
0000 0000 0000 1101
```

Wird dieser Wert mit `writeS` auf dem Bildschirm ausgegeben, so springt der Cursor einfach nochmals an den Zeilenanfang. Was ist passiert? Die beiden Bytes von `Lnge` werden einzeln auf dem Bildschirm dargestellt, und zwar zuerst das rechte Byte – also 13 bzw. Wagenrücklauf – dann das linke Byte – also 0 und damit Ende der Zeichenreihe. Eigentlich wollten wir aber die beiden Ziffern `'1'` und `'3'` ausgeben, und zwar als ASCII-Zeichen. Man muss also vor der Ausgabe den dualen Wert in eine Zeichenreihe umwandeln. Dies erledigt das folgende Makro.

`writeZ`	Ausgabe einer Dezimalzahl
Aufruf: `writeZ` *Zahl [, [Lnge] [, Ausgabe]]*	
Wirkung: • Es wird der Inhalt des 16 Bit-Wortes *Zahl* als Dezimalzahl ausgegeben. • *Lnge* gibt an, mit wie vielen Zeichen die Zahl rechtsbündig ausgegeben wird. Ist der Zahlenwert negativ, wird als erstes Zeichen ein Minus ausgegeben. **Voreinstellung** für *Lnge* ist 8. • *Ausgabe* bestimmt, wohin die Zahl ausgegeben wird (siehe Beschreibung von `print`).	
Bemerkung: • *Lnge* muss mindestens den Wert 3 haben. Wird *Lnge* kleiner gewählt, erscheint eine Laufzeit-Fehlermeldung. • Wird *Lnge* so klein gewählt, dass der Zahlenwert nicht dargestellt werden kann, wird die Ausgabe einfach von links her abgeschnitten.	

□ Beispiel 3.6.4

```
        DOSSEG
        .MODEL  SMALL
        .STACK  100H
include macros.mac
        .DATA
Hallo   db      10,13,10,13
        db      "*************************",10,13
Text    db      "*   Hallo, lieber Leser  *",10,13
        db      "*************************",10,13
        db      0       ; <<-- Text-Ende
Lnge    dw      ?
        .CODE
begin:  mov     ax,@Data
        mov     ds,ax

        print   "Bitte maximal 31 Zeichen eingeben : "
        readS   Text,31,Lnge
        writeS  Hallo
        print   "Sie haben "
        writeZ  Lnge,4
        println " Zeichen eingegeben"
```

```
        mov     ax,4c00h
        int     21h     ; Programm beenden
        end     begin   ; Ende des Programmtextes
```
■

Wenn Zahlen eingelesen werden, mit denen das Programm weiterrechnen soll, müssen diese von einer Zeichenreihe in eine Dualzahl umgewandelt werden. Dies erledigt – symmetrisch zu `writeZ` – das Makro `readZ`.

`readZ`	Einlesen einer Dezimalzahl
Aufruf: `readZ` *Zahl*	
Wirkung: • Über die Tastatur wird eine Zahl eingegeben und im 16 Bit-Wort *Zahl* als Dualzahl abgespeichert. Bemerkungen: • Maximal acht Zeichen – einschließlich Vorzeichen – werden berücksichtigt. • Wird ein Zeichen eingegeben, das keine Ziffer ist, wird es als Ziffer 0 interpretiert.	

Im folgenden Beispiel wird ein kleiner Taschenrechner realisiert, der zwei Zahlen addiert. Man kann auch negative Zahlen eingeben.

□ **Beispiel 3.6.5**

```
        DOSSEG
        .MODEL  SMALL
        .STACK  100H
include macros.mac
        zahl    dw      ?
        .CODE
begin:  mov     ax,@Data
        mov     ds,ax

        print   "Geben Sie den ersten Summanden ein : "
        readZ   ax
        print   "Geben Sie den zweiten Summanden ein: "
        readZ   zahl
        add     ax,zahl
        ;Addition der beiden eingelesenen Zahlen

        print   "Die Summe lautet -------->> : "
        writeZ  ax
        println
```

```
        mov     ax,4c00h
        int     21h
        end     begin
```

3.6.4 Zufallszahlen-Generator

In einigen Beispielen brauchen wir einen Zufallszahlen-Generator, der bei jedem Aufruf eine zufällige Zahl liefert. Damit nicht bei jedem Programmlauf dieselben „Zufallszahlen“ erzeugt werden, muss beim Start des Programms von irgendwo her ein zufälliger Wert berechnet werden. Das Makro `randomize` erledigt diese Aufgabe.

`randomize`	Zufallszahlen-Generator initialisieren
Aufruf: `randomize`	
Wirkung: • Dieses Makro initialisiert den Zufallszahlen-Generator.	

Das Makro `random` liefert bei jedem Aufruf eine zufällige ganze Zahl. Über die Parameter gibt man an, in welchem Bereich diese Zahlen liegen sollen. Bei der Programmierung eines Würfels wird man den Bereich zwischen 1 und 6 wählen.

`random`	nächste Zufallszahl erzeugen
Aufruf: `random`	*von, bis*
Wirkung: • Das Makro berechnet eine neue ganze Zufallszahl im Zahlenbereich zwischen *von* und *bis* – jeweils einschließlich – und legt sie im Register `dx` ab.	

Das folgende Programm gibt die Augenzahlen von zwei Würfen auf den Bildschirm aus.

☐ **Beispiel 3.6.6**

```
        DOSSEG
        .MODEL  SMALL
        .STACK  100H
include macros.mac
        .CODE
begin:  mov     ax,@Data
```