Assembler-Programmierung

W. Kippels

10. Februar 2015

Eine Anleitung mit Übungsaufgaben

Inhaltsverzeichnis

1 Vorwort 5

2 Wozu Assembler? 6

3 Die gewählte Umgebung 7
3.1 Der Betriebsmodus des Prozessors . . . 7
3.2 Die Entwicklungsumgebung . . . 7

4 Prozessorregister 8
4.1 Register des 8088 und 80286 . . . 8
4.1.1 Datenregister : . . . 8
4.1.2 Segmentregister: . . . 8
4.1.3 Adressregister: . . . 9
4.1.4 Flagregister: . . . 9
4.2 Register der Prozessoren ab xx386 . . . 10
4.3 Das Flag-Register . . . 11

5 Speicherverwaltung im Real-Modus 12

6 Speicherbelegung unter DOS 13

7 Aufbau einer COM-Datei 15

8 Grundstruktur von Assemblerprogrammen 16
8.1 Grundsätzliches . . . 16
8.2 Variablen . . . 17
8.3 Software-Interrupts . . . 18

9 Bedienung der Software 19
9.1 Der Assembler . . . 19
9.2 Der Debugger . . . 20

10 Das erste Programm 22

11 Sprungbefehle 23
11.1 Allgemeines zu Sprungbefehlen . . . 23
11.2 Bedingte Sprünge, Verzweigungen, Schleifen . . . 25

12 Übungsaufgaben, Teil 1: 26
12.1 Aufgabe 1.1 . . . 26
12.2 Aufgabe 1.2 . . . 26
12.3 Aufgabe 1.3 . . . 27
12.4 Aufgabe 1.4 . . . 27
12.5 Aufgabe 1.5 . . . 27

13 Stapelverarbeitung **27**

14 Übungsaufgaben, Teil 2 **29**
14.1 Aufgabe 2.1 29
14.2 Aufgabe 2.2 30

15 Unterprogramme **31**

16 Übungsaufgaben, Teil 3: **34**
16.1 Aufgabe 3.1 34
16.2 Aufgabe 3.2 35
16.3 Aufgabe 3.3 35

17 Zeiger und Segment-Override **37**
17.1 Zeiger 37
17.2 Segment-Override 39

18 Übungsaufgaben, Teil 4: **43**
18.1 Aufgabe 4.1 43
18.2 Aufgabe 4.2 43
18.3 Aufgabe 4.3 44
18.4 Aufgabe 4.4 45

19 Diverses **47**
19.1 Hardwarenahe Befehle 47
19.2 Interrupts umleiten 47

20 Übungsaufgaben, Teil 5 **53**
20.1 Aufgabe 5.1 53
20.2 Aufgabe 5.2 59

21 Einbinden von Assembler in Hochsprachen **64**

22 Übungsaufgaben, Teil 6: **72**
22.1 Aufgabe 6.1 72

23 Musterlösungen, Teil 1 **74**
23.1 Aufgabe 1.1 (ABFRAGE) 74
23.2 Aufgabe 1.2 (ABFRAGE1) 74
23.3 Aufgabe 1.3 (ABFRAGE2) 75
23.4 Aufgabe 1.4 (10TASTEN) 76
23.5 Aufgabe 1.5 (ZIFFER) 79

24 Musterlösungen, Teil 2: **81**
24.1 Aufgabe 2.1 (ZEICHEN) 81
24.2 Aufgabe 2.2 (TASTEN) 82

25 Musterlösungen, Teil 3: **84**
25.1 Aufgabe 3.1 (ZIFFERN) 84
25.2 Aufgabe 3.2 (CURSOR) 86
25.3 Aufgabe 3.3a (HTASTEN) 91
25.4 Aufgabe 3.3b (HTASTEN2) 94

26 Musterlösungen, Teil 4: **96**
26.1 Aufgabe 4.1a (BLINKEIN) 96
26.2 Aufgabe 4.1b (BLINKAUS) 97
26.3 Aufgabe 4.2 (HALLO) 98
26.4 Aufgabe 4.3a (DTASTE) 102
26.5 Aufgabe 4.3b (DTASTE2) 106
26.6 Aufgabe 4.4 (TPUFFER) 109

27 Musterlösungen, Teil 5 **113**
27.1 Aufgabe 5.1 (TON) 113
27.2 Aufgabe 5.2 (SCANCODE) 119

28 Musterlösungen, Teil 6: **122**
28.1 Aufgabe 6.1 (ASTEST) 122

1 Vorwort

Dieser Assembler-Kurs soll einen Einstieg in die Assembler-Programmierung im Real-Mode unter DOS vermitteln. Mit zu diesem Lehrgang gehören noch folgende Dateien:

- Die Befehlsliste: http://dk4ek.de/lib/exe/fetch.php/befehle.pdf
- Die BIOS-Interrupts: http://dk4ek.de/lib/exe/fetch.php/bios_int.pdf
- Die DOS-Funktionen: http://dk4ek.de/lib/exe/fetch.php/dos_int.pdf

Unter den angegebenen Links sind diese Dateien erhältlich. Ohne Kenntnisse dieser Daten ist eine Assembler-Programmierung kaum möglich.

An Software werden benötigt:

- Der Assembler A86
- Der zugehörige Debugger D86
- Ein beliebiger DOS-Editor

Die Programme A86/D86 sind Shareware. Zum Testen können diese Programme frei verwendet werden. Wer sich ersthaft länger mit diesen Programmen beschäftigen will, der muss sich als Nutzer registrieren lassen. Dafür bekommt er dann die Voll-Versionen dieser Programme, die A386 und D386 heißen. Im Gegensatz zu A86 und D86 unterstützen diese Versionen nicht nur den Befehlssatz und die Register des 8088/80286, sondern auch die erweiterten Register ab 80386 sowie den zugehörigen Befehlssatz. Zum Testen kann man aber sicher problemlos mit A86 und D86 arbeiten. Diese Programme sind hier erhältlich:
http://eji.com/a86

Als DOS-Editor empfehle ich EDDI. Er ist frei zu verwenden. Ich habe ihn zu etwa 80 Prozent in Assembler geschrieben, der Rest ist Pascal. Er ist hier erhältlich:
http://dk4ek.de/lib/exe/fetch.php/eddi.zip

Zum Programmieren wird eine DOS-Umgebung benötigt. Unter Linux kann das das Programm-Paket „DOSEMU“ sein zusammen mit „FREEDOS“. Beides ist in fast jeder Distribution verfügbar. Alternativ kann auch das Programm-Paket „DOSBOX“ verwendet werden. Ich habe beides ausprobiert, beides ist empfehlenswert. Wer unter Windows arbeiten möchte, kann direkt die Kommandozeile (Eingabeaufforderung) nutzen. Allerdings funktionieren einzelne BIOS-Funktionen nicht (beispielsweise die Funktion 83h des Interrupt 15h), da sie von Windows nur unvollständig emuliert werden. Sollten Sie Töne erzeugen wollen, kann es nötig sein, mit Alt-Enter auf den Vollbildmodus umzuschalten. Wenn Sie mit einem 64-Bit-System arbeiten, dann hilft nur noch das Programm „DOSBOX“ weiter. Das gibt es auch (gratis) für Windows, allerdings weiß ich nicht wo. Googeln hilft sicher schnell weiter.

2 Wozu Assembler?

Assembler ist die einzige Programmiersrache, die sich 1:1 in Maschinensprache (und zurück) übersetzen lässt. Jedem Assembler-Befehl entspricht ein Maschinencode-Befehl. Theoretisch könnte man also auch direkt Maschinencode schreiben. Dieser ist jedoch für den Menschen reichlich unlesbar. Ein Beispiel: Der Maschinencode-Befehl `010BBA` lädt die Zahl 10Bhex in das Register DX. Unter Assembler liest sich der Befehl so: `mov DX,010B` Das ist für uns Menschen besser zu verstehen. Dennoch bildet Assembler den Maschinencode direkt ab.

Weil das so ist, kann man jedes Programm, das mit einer beliebigen Programmiersprache erstellt wurde, im Debugger so betrachten, dass man die Assembler-Befehle sieht. Aus diesem Grund kann Assembler durchaus als Grundlage für alle Hochsprachen angesehen werden. Da die Sprache sehr hardwarenah ist – beim Programmieren werden direkt die „Innereien“ des Prozessors angesprochen – kann man bei der Beschäftigung mit Assembler quasi nebenbei vieles über die Funktionsweise des Rechners erfahren. Darin sehe ich den wichtigsten Aspekt für die Beschäftigung mit der Assembler-Programmierung.

Niemand würde heute ernsthaft ein komplettes PC-Programm in Assembler schreiben. Das wäre viel zu aufwändig. Was sich aber schon lohnen kann, ist das Schreiben einer Assembler-Routine, die man in die Hochsprache einbindet. Die kann dann auf Ablaufgschwindigkeit hin optimiert werden. Weiterhin sind Assemblerkenntnisse erforderlich, wenn es darum geht, einen unbekannten Virus zu untersuchen. Hier wird üblicherweise kein Quellcode mitgeliefert. Dann gibt es natürlich auch noch Sonderanwendungen wie etwa ein Bootloader. Dieser muss (bekanntlich) im 512-Byte-kleinen Bootsektor Platz finden. Auch dafür bietet sich Assembler an, denn damit kann man sehr kleine ausführbare Dateien erzeugen.

Die meisten Prozessoren findet man übrigens nicht im PC, sondern in allerlei Geräten des täglichen Lebens. Sei es das Mobiltelefon, die Armbanduhr, der MP3-Spieler oder der Fahrradtacho – hier haben die Prozessoren wegen Batteriebetrieb eine sehr niedrige Taktrate, und auch der Speicherplatz ist oft recht klein. Daher wird auch hier immer noch oft Assembler-Programmierung eingesetzt. Allerdings muss man beachten, dass jeder Prozessor seine eigene Assembler-Sprache hat.

3 Die gewählte Umgebung

3.1 Der Betriebsmodus des Prozessors

Die heute gängigen PC-Prozessoren kennen zwei Betriebsmodi: den „Real Mode“ und den „Protected Mode“. Diese haben historische Hintergründe. Als Anfang der 80-er Jahre der 8088-Prozessor entwickelt wurde, sollte ein Rechner entstehen, der jeweils nur eine einzige Aufgabe zu erledigen hat. An Multitasking hat damals noch niemand gedacht. Der Prozessor kannte nur den „Real Mode“, in dem dem abzuarbeitenden Programm keinerlei Einschränkungen gemacht wurden. Erst später – bei der Einführung des 80386-Prozessors – hatte man die Notwendigkeit der Abschottung mehrerer gleichzeitig laufender Prozesse gegeneinander erkannt und als „Protected Mode“ eingeführt. Aus Kompatibilitätsgründen blieb aber der Real Mode als zusätzliche Betriebsart erhalten. In diesem Modus beginnt übrigens der Bootvorgang des Rechner auch heute noch. Moderne Betriebssysteme wie Unix, Linux und auch Windows arbeiten heute im Protected Mode.

In diesem Kurs soll es darum gehen, einen möglichst schnellen einfachen Einstieg in die Assembler-Programmierung zu bekommen. Da im Protected Mode allerlei zusätzliche Dinge zu beachten sind, wird dieser Kurs den Prozessor im Real Mode betreiben. Das funktioniert eigentlich nur mit dem Betriebssystem DOS. Es funktioniert aber auch im DOS-Modus unter Windows (Kommandozeile) oder im DOS-Emulator unter Linux. Hier arbeitet der Prozessor dann zwar in einem „Virtuellen Real Mode“, indem er in einem Fenster oder auf einer Konsole den Real Mode simuliert. Für uns als Programmierer stellt sich dabei aber der Prozessor so dar, als ob er im Real Mode arbeiten würde. Alle Einschränkungen und Besonderheiten des Protected Mode brauchen also nicht beachtet zu werden.

3.2 Die Entwicklungsumgebung

Als Werkzeuge verwende ich den Shareware-Assembler A386 und den zugehörigen Debugger D386. Diese sind hier erhältlich: http://eji.com/a86

Die meisten Assembler (so nennt man den Compiler) können nur Objekt-Code erzeugen, der dann anschließend mit einem Linker zu einem lauffähigen Programm „gelinkt“ werden muss. Der A386 (und auch die abgespeckte offene Version A86) kann dagegen direkt eine lauffähige COM-Datei erzeugen. Das ist die einfachste denkbare Form einer ausführbaren Datei unter DOS. Weiterhin erspart uns dieser Assembler allerlei sonstige zusätzliche Angaben im Quelltext, die dann der Linker benötigt. Daher können wir mit diesem Assembler recht schnell und einfach Programme erzeugen. Außerdem wird beim Compilieren gleichzeitig eine Symbol-Datei erzeugt, mit deren Hilfe der zugehörige Debugger D386 (und auch die abgespeckte offene Version D86) recht komfortabel die im Quelltext verwendeten Variablennamen und Sprungmarken an der entsprechenden Stelle einblendet.

4 Prozessorregister

Um sinnvoll mit dem Programmieren beginnen zu können, muss man die Register der Prozessoren kennen, denn diese werden bei fast allen Befehlen direkt angesprochen. Im Laufe der Jahre wurden diese ergänzt und auch erweitert. Beginnen wir daher beim Aufbau der ersten Prozessoren.

4.1 Register des 8088 und 80286

Alle Register des 8088 und 80286 haben Word-Format, also 16-Bit Breite. Einige Register sind auch partiell als Byte oder Bit ansprechbar. (Ab xx386 siehe nächstes Kapitel.)

Die Register haben unterschiedliche Aufgaben. Einige können ausschließlich für bestimmte Funktionen verwendet werden, andere hingegen können mehr oder weniger beliebig verwendet werden.

4.1.1 Datenregister :

Alle Datenregister **AX**, **BX**, **CX**, **DX** können beliebig für fast alle Operationen verwendet werden, haben aber unterschiedliche Aufgaben bei einzelnen speziellen Operationen. Alle Datenregister können auch (mit dem zweiten Buchstaben L für low oder H für high) byteweise angesprochen werden. Im einzelnen sind das:

- **AX (aufteilbar in AL und AH): Akkumulator**. AX wird als häufigstes allgemeines Datenregister verwendet. Es hat eine besondere Bedeutung bei Rechenoperationen sowie bei Stringoperationen mit Zeigern.
- **BX (aufteilbar in BL und BH): Basis**. BX hat eine besondere Bedeutung als Hilfszeiger.
- **CX (aufteilbar in CL und CH): Zähler**. Seine besondere Bedeutung ist als Zähler bei Schleifen und anderen mehrfach auszufühernden Operationen.
- **DX (aufteilbar in DL und DH): Daten**. DX wird zusammen mit AX für bestimmte Rechenoperationen verwendet.

4.1.2 Segmentregister:

Alle Daten wie Programmcode, Daten oder der Stapel liegen in einem bestimmten Bereich, dem jeweiligen Segment. Die Adresse des jeweiligen Segmentes steht in einem Segmentregister. Bei der Adressierung des Speichers durch den Prozessor benötigt dieser **immer** die gültige Segmentadresse in dem jeweiligen Segmentregister. Daher können diese (bis auf ES) **nicht** für andere Zwecke verwendet werden. Hier ihre Namen und ihre Aufgaben:

- **CS: Codesegment**; hier steht der Programmcode

- **DS: Datensegment**; hier liegen die Variablen
- **SS: Stapelsegment**; hier liegen die Daten des Stapels
- **ES: Extrasegment**; Zielsegment bei Stringoperationen mit Zeigern

4.1.3 Adressregister:

Alle Adressregister können (ggf. mit zusätzlichem Einsatz von BX) als Zeiger zur Adressierung verwendet werden. Sie sind **nicht** byteweise ansprechbar, also nicht in High- und Low-Byte aufteilbar, wie die Register AX bis DX. Bei Bedarf können sie aber auch für viele Funktionen als allgemeines Register verwendet werden. Hier ihre Namen und ihre Aufgaben:

- **SP: Stapelzeiger**; zeigt auf nächsten freien Platz auf dem Stapel im SS. SP kann **ausschließlich** für diesen Zweck verwendet werden.
- **BP: Basiszeiger**; wird zur Verwaltung lokaler Variablen auf dem Stapel SS verwendet. Alle Hochsprachen machen das so. Daher sollte man BP nicht anders verwenden, auch wenn das im Prinzip möglich wäre.
- **SI: Quellzeiger**; zeigt auf Quell-Variable bei Stringoperationen in DS. Solange nicht gerade eine der angesprochenen String-Operationen im Gange ist, kann SI ohne weiteres auch für andere Zwecke verwendet werden.
- **DI: Zielzeiger**; zeigt auf Ziel-Variable bei Stringoperationen in ES, ansonsten DS. DI kann wie SI auch anders verwendet werden, wenn man das möchte.
- **IP: Befehlszeiger**; er zeigt immer auf den nächsten auszuführenden Befehl, er kann nicht willkürlich verändert werden. Daher ist es auch ausgeschlossen, IP für andere Zwecke zu verwenden.

4.1.4 Flagregister:

Normalerweise werden die Flags nur bitweise angesprochen. Sie kennzeichnen bestimmte Zustände oder Ergebnisse bestimmter Operationen. Mit den Befehlen pushf und popf können aber alle Flags gleichzeitig auf dem Stapel abgelegt bzw. vom Stapel zurückgeholt werden. Die Zustände der Flags bestimmen das Verhalten des Programms bei Abfragen. Hier die Bits des Flagregisters, auf deren Bedeutung etwas später eingegangen wird:

-	-	-	-	OF	DF	IF	TF	SF	ZF	-	AF	-	PF	-	CF

4.2 Register der Prozessoren ab xx386

Ab der Prozessorfamilie xx386 bis zum Pentium sind die Daten- und Adress-Register auf auf die Länge von 32 Bit (Double-Word) erweitert worden. Auch das Flagregister wurde auf die gleiche Größe erweitert. Die erweiterten Register werden mit einem vorangestellten E angesprochen. Die alten Namen bleiben für die untere Hälfte (Word) erhalten. Die Register **AX**, **BX**, **CX** und **DX** stellen somit das untere Word der Register **EAX**, **EBX**, **ECX** und **EDX** dar und können nach wie vor einzeln angesprochen werden, also auch **AH** und **AL** einzeln. Es ist jedoch nicht vorgesehen, auch das obere Word einzeln ansprechen zu können. Die erweiterten Adress-Register heißen entsprechend **ESP**, **EBP**, **ESI** und **EDI** und enthalten die Register **SP**, **BP**, **SI** und **DI** als unteres Word. Ab dem 80386 können alle sieben 32-Bit-Register (**EAX**, **EBX**, **ECX**, **EDX**, **ESI**, **EDI** und **EBP**) im Protected Mode als Hilfsregister für höhere Adressierungsarten verwendet werden.

Die Segment-Register **CS**, **DS**, **SS** und **ES** sind unverändert (mit Word-Länge) erhalten geblieben, jedoch sind noch zwei weitere Extra-Segment-Register hinzugekommen, nämlich **FS** und **GS**. Es gibt also keine Segmentregister in Doppelwort-Länge.

Auch zum Befehlszeiger **IP** (Instruction-Pointer) gibt es jetzt das erweiterte Gegenstück, den **EIP**. Allerdings können weder **IP** noch **EIP** direkt beeinflusst werden. Zudem steht der Zeiger **EIP** nur im Protected Mode zur Verfügung. Wie bereits erwähnt, laufen alle modernen Betriebssysteme wie Linux, OS/2 und auch Windows im Protected Mode, nicht aber DOS. Hier ist der Prozessor in den Real Mode geschaltet. Das hat den angenehmen Nebeneffekt, dass das Schreiben eines Assembler-Programms unter DOS einfacher ist.

Gleichfalls bei den Flags gibt es eine Erweiterung. Mit **EFLAGS** wird ein Doppelwort mit den Flags angesprochen. Im erweiterten Bereich sind bisher jedoch erst 6 Flags definiert, auf die hier aber nicht weiter eingegangen werden soll.

4.3 Das Flag-Register

Das Flag-Register besteht aus 16 Bits, die zwar zu einem word zusammengefasst sind, die aber normalerweise einzeln beeinflusst oder geprüft werden. Hier die Bits des Flagregisters:

-	-	-	-	OF	DF	IF	TF	SF	ZF	-	AF	-	PF	-	CF

Nicht alle Bits sind (derzeit) belegt. Einige Flags können per Befehl gesetzt oder gelöscht werden. Hier die Bedeutung der Flags im einzelnen:

OF=Overflow- oder Überlauf-Flag Das OF wird gesetzt, wenn beim Rechnen mit vorzeichenbehafteten Zahlen ein Überlauf auftritt, genauer: wenn vom Bit 6 nach 7 (bei Byte-Operanden) oder Bit 14 nach 15 (bei Word-Operanden) ein Übertrag entsteht und nicht gleichzeitig ein Überlauf aus dem Byte oder Word hinaus (dann wird das CF gesetzt, siehe unten).

DF=Direction- oder Richtungs-Flag Das DF wird per Befehl gesetzt oder gelöscht (**std** bzw. **cld**). Es legt bei String-Operationen fest, ob die Zeichenkette in auf- oder absteigender Richtung durchlaufen wird. 0 (oder „nicht gesetzt“) bedeutet, es wird in aufsteigender Richtung gearbeitet.

IF=Interrupt-Flag Das IF kann per Befehl gesetzt oder gelöscht werden (**sti** bzw. **cli**). Ist es gesetzt, lässt der Prozessor Hardware-Interrupts zu.

TF=Trap- oder Einzelschritt-Flag Ist das TF gesetzt, geht der Prozessor in den Einzelschrittmodus (nur von Bedeutung für Debugger-Programme).

SF=Sign- oder Vorzeichen-Flag Das SF ist identisch mit dem höchstwertigen Bit des Ergebnisses bei Operationen.

ZF=Zero- oder Null-Flag Das ZF wird gesetzt, wenn das Ergebnis der letzten Operation Null ergab.

AF=Auxiliary-Carry- oder Hilfsübertrags-Flag Das AF zeigt einen Übertrag von Bit 3 nach Bit 4 nach einer Addition oder Sutraktion an.

PF=Parity- oder Paritäts-Flag Das PF wird gesetzt, wenn das niederwertige Byte des Ergebnisses eine gerade Anzahl von Einsen enthält. (Heute von geringer Bedeutung)

CF=Carry- oder Übertrags-Flag Das CF wird gesetzt, wenn ein Übertrag aus dem höchstwertigen Bit heraus (bei Addition) oder in das höchtwertige Bit hinein (bei Subtraktion) – auch Borge-Übertrag genannt – erfolgt ist. Das CF kann auch per Befehl gesetzt oder gelöscht werden (**stc** bzw. **clc**).

Welche Befehle im einzelnen welche Flags beeinflussen, finden Sie bei der Beschreibung der Befehle. Befehle, die nur Daten bewegen, also keine Rechenoperationen im weiteren Sinne sind, beeinflussen keine Flags.

5 Speicherverwaltung im Real-Modus

Die Prozessoren der x86-Familie kennen zwei Betriebsmodi: den Real Mode und den Protected Mode. Diese Modi sind historisch entstanden. Der Ur-PC mit dem 8088 konnte nur 1 MB Arbeitsspeicher adressieren, dazu wurde der Real Mode geschaffen. Mit Einführung des 80286 wurde als zusätzlicher Modus der Protected Mode eingeführt, womit mehr als 1 MB adressiert werden können. DOS und auch ältere Windows-Versionen (bis Version 3.11) arbeiten (fast) ausschließlich im Real Mode und auch alle Betriebssystem-Funktionen im BIOS heutiger PCs sind hierfür programmiert. Wir werden in unserem Kurs ausschließlich im Real Mode arbeiten.

Um 1 MB zu adressieren sind 20 Steuerleitungen notwendig, denn 1 MB sind 2^{20} Byte. Die **physikalische** Adresse ist also 20 Bit lang, zu lang also für ein Word-Register mit 16 Bit. Man benötigt also 2 Register. Damit könnte man zwar bis zu 4 GB (2^{32} Bit) adressieren, dazu sah aber bei der Einführung des Ur-PC niemand auch nur ansatzweise eine Notwendigkeit. Bei älteren Betriebssystemen wie CPM (vor DOS) musste das Programm grundsätzlich immer an der gleichen Stelle im Speicher geladen werden, weil jedes Programm Verzweigungen zu bestimmten Adressen enthält. Es war also nicht möglich, mehrere Programme (wie zum Beispiel einen Tastaturtreiber neben dem Hauptprogramm) gleichzeitig ablaufen zu lassen. Wie wir gleich sehen werden, ist dies beim Speicherkonzept im Real Mode nun gut möglich. Man hat nun die (im Hexcode fünfstellige) Adresse **überlappend** in je eine (im Hexcode vierstellige) Segment- und Offset-Adresse zerlegt. Im Prozessor werden die beiden Werte dann zur Bildung der physikalischen Adresse um vier Bit verschoben addiert. Ein Beispiel: Eine physikalische Adresse sei 35A7F. Nachfolgend sind mehrere Möglichkeiten gezeigt, wie diese physikalische Adresse zerlegt werden kann.

Segment:	35A7	35A0	3517	3123
Offset:	000F	007F	090F	484F
Summe:	35A7F	35A7F	35A7F	35A7F

Es gibt also keine eindeutige Zerlegung einer physikalischen Adresse. Das hat zur Folge, dass ein Programm an jeder durch 16 teilbaren physikalischen Adresse (letzte Hex-Ziffer=0) geladen werden kann, wenn innerhalb des Programms nur die Offset-Adresse bei Sprüngen und Verzweigungen verwendet wird. Der Segment-Anteil der jeweiligen

Adresse steht dann einfach in einem bestimmten **Segmentregister**. Beim Start des Programms stellt das Betriebssystem diese Segmentregister auf den richtigen Wert ein und schon kann das Programm gestartet werden.[1] Der Befehlszeiger **IP** beispielsweise bezieht sich dann immer auf das Segment-Register **CS**; das bedeutet, dass die physikalische Adresse entsprechend obigem Beispiel aus den Inhalten von **CS** und **IP** gebildet wird. Die komplette Adresse wird dann so dargestellt: **CS:IP** . Man schreibt vorn die Segment-Adresse hin, danach einen Doppelpunkt und dahinter die Offset-Adresse, also auch beispielsweise: 35A0:007F .

In einem Programm gibt es üblicherweise 3 Segmente, nämlich das bereits angesprochene Codesegment **CS**, in dem der Programmcode liegt, das Datensegment **DS**, in dem die Daten liegen, und das Stack-Segment **SS**, in dem der für vielerlei Zwecke verwendete Stapel liegt. Bei allen Daten-bezogenen Befehlen wie **MOV**, **CMP**, **ADD** usw. wird automatisch der Inhalt von **DS** für die Adressenbildung verwendet, bei Stapel-bezogenen Befehlen wie **PUSH**, **POP**, usw. und Operationen mit dem Zeiger **BP**, der Inhalt von **SS** und bei Code-bezogenen Befehlen wie **JMP**, **CALL** usw. entsprechend der Inhalt von **CS**. Einzelheiten dazu folgen später.

In einem Programm mit der Endung *.COM liegen alle Daten im **gleichen** Segment, also stellt DOS die Werte von **CS**, **DS**, **SS** und **ES** beim Start auf den gleichen Wert ein. Wir werden uns zunächst mit solchen COM-Programmen beschäftigen und brauchen uns daher im Anfang um die Segment-Adressen keine weiteren Sorgen zu machen.

6 Speicherbelegung unter DOS

Wie bereits angesprochen, können unter DOS (normalerweise) 1024 KB angesprochen werden, denn DOS arbeitet im Real Mode. Trotzdem spricht man nur von maximal 640 KB unter DOS. Wieso? Nachfolgend ist etwas vereinfacht die Verwendung des Speichers unter DOS dargestellt. Ab Adresse A0000 wird der Speicher für die Hardware und das BIOS benötigt. Es gibt zwar ggf. noch Lücken je nach PC, die stehen aber zunächst einmal nicht zur Verfügung. Der Bereich bis 9FFFF sind die erwähnten 640 KB.

Speicherbelegung unter DOS:

C8000-FFFFF	BIOS
A0000-C7FFF	Grafikkarte
00600-9FFFF	DOS-Programme und Treiber
00500-005FF	DOS-Datenbereich
00400-004FF	BIOS-Datenbereich
00000-003FF	Interrupt-Tabelle

[1] Genau so funktioniert das COM-Modell unter DOS.

Nehmen wir mal an, der Bereich bis 1C32F ist belegt durch diverse Treiber (und natürlich die markierten Bereiche am Speicheranfang), dann könnte unser Programm ab der Adresse 1C330 in den Speicher geladen werden. Nehmen wir weiterhin zur Vereinfachung an, wir starten ein COM-Programm, dann geschieht das Starten des Programms dadurch, dass alle Segmentregister auf 1C33 gesetzt werden. Ab Offset 0 steht der Speicher also unserem Programm zur Verfügung. Dort beginnt allerdings nicht der Code unseres Programmes, sondern von Offset 0000 bis 00FF liegt der PSP-Bereich, den jedes Programm benötigt. Dort stehen einige wichtige Daten drin wie beispielsweise die Rücksprungadresse nach Beendigung des Programms und von 0080 bis 00FF der DTA-Bereich, in dem beispielsweise beim Programmstart ein Übergabestring abgelegt wird. Ab der Adresse 0100 startet das Programm. Der zugehörige Stapelbereich liegt am oberen Ende des Segmentes, genauer: der Stackpointer SP wird von DOS auf FFFE initialisiert. Jedes mal, wenn etwas „auf den Stapel gelegt“ wird, verringert sich SP um 2, und dann wird der Wert da hin geschrieben wo SS:SP hinzeigt. Der Stapel „wächst“ somit quasi nach unten, also dem Programmcode und den Programmdaten entgegen. Es ist nun Sache des Programmierers, darauf zu achten, dass der verfügbare Stapelbereich groß genug ist, denn sonst werden gnadenlos andere Daten oder Programmcode überschrieben. Nebenbei bemerkt sind etliche Sicherheitslücken in diversen WINDOWS-Versionen auf solche Stapel-Überläufe (Stack-Overflows) zurückzuführen.

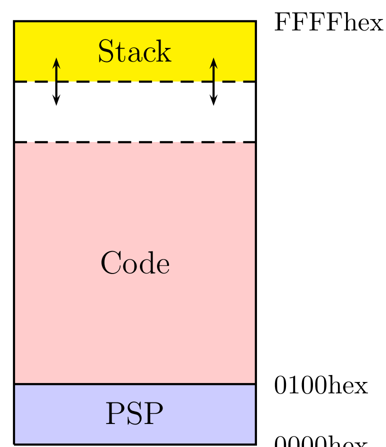
In höheren Programmiersprachen wird für Code, Daten und auch den Stack ein eigener Bereich mit einer eigenen Segment-Adresse reserviert. Ist ein solcher Bereich kleiner als 64 KB, dann kann trotzdem bei einer Bereichsüberschreitung ein Überschreiben „in fremdem Gebiet“ passieren. In Assembler ist es uns völlig freigestellt, wo wir Daten und wo wir Programmcode ablegen wollen. Es gibt lediglich einige Empfehlungen, wie man es machen kann. Auf die wird später noch eingegangen.

Was ich noch verschwiegen habe, ist für uns von untergeordneter Bedeutung. So wird beispielsweise vom Betriebssystem für jedes Programm noch ein Environment-Bereich angelegt. Dort stehen einige wichtige Informationen wie etwa der Suchpfad drin. Wo der Environment-Bereich zu finden ist, steht wiederum im PSP des Programms. Weiterhin legt DOS zu jedem Speicherblock, der einem Programm zugeteilt wurde, eine kleine Tabelle über die Größe und die Lage an. Die genauen Einzelheiten dazu würden allerdings den Rahmen dieser Anleitung sprengen.

7 Aufbau einer COM-Datei

Da wir uns in unserem Kurs (fast) ausschließlich mit COM-Dateien beschäftigen, lohnt es sich, deren Aufbau etwas genauer anzusehen. Das wichtigste ist: **Das gesamte Programm mit allem „Zubehör" befindet sich in *einem einzigen* Speichersegment.** Das bedeutet, dass **alle** benutzten Segmentregister (**DS, CS, SS, ES**) auf das **gleiche** Segment eingestellt sind. Wir brauchen uns darum also nicht zu kümmern. Das macht es dem Anfänger etwas einfacher. Nur die **Offset-Adresse** ist von Belang.

Von Offset-Adresse 0000hex bis 00FFhex befindet sich der PSP-Bereich, der uns zunächst auch nicht weiter interessiert. Ab Offset-Adresse 0100hex befindet sich der ausführbare Code im Speicher. Das bedeutet, dass die Adresse 0100hex **immer** die Startadresse des Programms ist. Bis zu welcher Adresse der Programmcode reicht, ist logischerweise von der Programmgröße abhängig.[2]



Aufbau einer COM-*Datei*

Der Bereich am Segmentende, also unterhalb bis einschließlich Offset-Adresse FFFFhex wird für den Stapel (den Stack) verwendet. Wird etwas auf dem Stapel abgelegt, „wächst" der Stapel nach unten. Passt man hierbei nicht auf, ist es durchaus denkbar, dass er bis in den Bereich hineinwächst, in dem gültiger ausführbarer Code steht. **Im Betriebssystem gibt es dagegen *keinen* Schutz!** Der Programmierer hat also selbst darauf zu achten, dass das nicht passieren kann. Gemeinerweise benutzen gelegentlich auch „fremde" Routinen unseren Stapel mit. Hierbei handelt es sich in der Regel um Interrupt-Routinen vom BIOS oder von DOS. Daher sollte man noch eine Reserve von mindestens 30 bis 50 Byte bereithalten.

In unseren Übungsprogrammen sollte uns diese Problematik allerdings keine Kopfschmerzen machen. Unsere Übungsprogramme sind alle so winzig klein, dass wir immer ausreichend Reserve haben. Bei fehlerhaften Programmen kann es aber vorkommen, dass immer mehr auf dem Stapel abgelegt wird, ohne je wieder entfernt zu werden. Dann läuft er irgendwann über und es treten sonderbare und kaum reproduzierbare Effekte auf.

Interessant ist noch die obere Hälfte des PSP-Bereiches, also der Bereich von 007Fhex bis 00FFhex. In diesem Bereich legt DOS den Übergabestring ab. Wird also ein Programm mit Übergabeparametern an der Kommandozeile gestartet, etwa `format c:`, dann wird der Text `c:` hier zur weiteren Bearbeitung bereitgelegt.

[2]Da der Code in ein einziges Segment passen muss, können COM-Dateien auch nie größer als (knapp) 64 kB sein.

8 Grundstruktur von Assemblerprogrammen

8.1 Grundsätzliches

Jeder Befehl belegt genau eine Programmzeile. Eine Leerzeile wird nicht ausgewertet. Leerzeichen dürfen nach Belieben vor und hinter einem Befehl stehen. Die Befehle haben keinen, einen oder zwei zusätzliche Parameter, je nach Art des Befehles. Zwischen dem Befehl und dem ersten Parameter steht (mindestens) ein Leerzeichen, der zweite Parameter wird mit einem Komma an den ersten angehängt. Alles kann in Groß- oder Kleinbuchstaben geschrieben werden. Zahlen werden als Dezimalzahlen interpretiert. Zahlen im Hexadezimalformat werden durch ein angehängtes „h“ oder „hex“ gekennzeichnet. Der Assembler A386 erkennt auch Zahlen mit einer führenden 0 als Hexadezimalzahlen. (Das ist sogar seine bevorzugte Schreibweise.) Also ist 21h = 21hex = 021 = 33.

Beispiele:

```
ret             ; Rücksprung aus einem Unterprogramm
pushf           ; Lege den Inhalt des Flag-Registers auf den Stapel
popf            ; Hole den Inhalt des Flag-Registers vom Stapel zurück
int  021        ; Löse den Interrupt mit der Nummer 21hex aus
push AX         ; Lege den Inhalt des Registers AX auf den Stapel
pop  AX         ; Hole den Inhalt des Registers AX vom Stapel zurück
mov  AX,BX      ; Schreibe den Inhalt des Registers BX ins Register AX
mov  AX,300     ; Schreibe die Zahl 300 ins Register AX
mov  AX,W[300]  ; Schreibe die Word-Variable nach AX, die an der Offset-Adresse
                ; 300 steht
```

Alle Zeichen hinter einem Semikolon werden vom Compiler nicht beachtet; dort werden Kommentare zum Programm geschrieben.

Einefeste Struktur im Programm wie etwa bei PASCAL gibt es nicht. Variablen können am Anfang, am Ende oder auch mitten im Programm deklariert werden. Aber aufgepaßt: Genau dort, wo sie deklariert werden, werden sie dann auch angelegt! Man kann jedoch auch explitit bei der Variablendeklaration mit der Direktrive **SEGMENT** und dem Befehl **org** einen Bereich (Startadresse) angeben, wo die Variablen angelegt werden sollen.

Beispiel:

```
DATA SEGMENT  ; Der Bereich der Variablen soll beginnen...
  org  0A000  ; ...an der Stelle A000 hexadezimal
  Var1 DB ?   ; Variable Var1 sei eine Byte-Variable ohne festen Startwert
  Var2 DB ?   ; Variable Var2 sei eine Byte-Variable ohne festen Startwert
  Var3 DW ?   ; Variable Var3 sei eine Word-Variable ohne festen Startwert
DATA ENDS     ; Hiermit endet die Variablendeklaration im vorgegebenen Bereich
              ; Anschließend beginnt der Programmcode
```

Eine andere Möglichkeit für eine Deklaration im Programmcode ermöglicht es, die definierten Variablen mit Startwerten vorzubelegen.

Beispiel:

```
jmp   START    ; Da hier das Programm beginnt: Sprung zum Label START
Var1  DB 15    ; Variable Var1: Byte-Variable mit Startwert 15
Var2  DB 'a'   ; Variable Var2: Byte-Variable mit ASCII-Zeichen a als Startwert
Var3  DW 020   ; Variable Var3: Word-Variable mit Startwert 32 (=20h)
START:         ; Hier beginnt der eigentliche Programmcode
```

8.2 Variablen

Variablen werden bei ihrer Deklaration nur unterschieden nach ihrer **Länge**, nicht aber nach ihrer **Bedeutung**. Die Deklaration besteht aus den 3 Bestandteilen: *Name*, *Typ* und *Wert*. Dabei kann bei Bedarf der Name entfallen; dies ist allerdings nur in Sonderfällen sinnvoll.

Mit folgenden Befehlen werden folgende Typen definiert:

Befehl:	DB	DW	DD	DQ	DT
Typ:	Byte	Word	Double-Word	Quad-Word	Ten-Bytes
Länge:	(1 Byte)	(2 Byte)	(4 Byte)	(8 Byte)	(10 Byte)

Mit *Wert* wird der Startwert der Variablen festgelegt. Steht dort ein Fragezeichen „?“, dann erfolgt keine Initialisierung, der Startwert ist also ein Zufallswert. ASCII-Zeichen belegen bekanntlich ein Byte. Will man die Byte-Variable `BUCHSTABE` mit einem „A“ initialisieren, dann kann man natürlich eine 65 oder 41hex angeben; das ist der numerische Wert des ASCII-Zeichens „A“. Bequemer geht es mit einem ASCII-Zeichen in einfachen Anführungszeichen, also beispielsweise so:
`BUCHSTABE  DB 65` oder eben übersichtlicher: `BUCHSTABE  DB 'A'`

Es können auch mehrere gleichartige Variablen zu einem **Feld** zusammengafaßt werden. Dabei kann die Feldlänge mit dem Schlüsselwort `DUP` und einer davor(!) stehenden Zahl angegeben werden. Der Wert steht dann dahinter. Statt mit `DUP` kann die Feldlänge auch durch eine Anzahl unterschiedlicher mit Komma getrennter Initialisierungswerte angegeben werden. Ist das Feld ein String, dann können alle ASCII-Zeichen in Hochkomma eingeschlossen werden.

Beispiele:

`Zahl    DB 15`	Byte-Variable initialisiert mit 15 (=0Fh)
`Zeichen DB 'a'`	Byte-Variable initialisiert mit 97 (=61h='a')
`Feld1   DB 1,3,5,7`	4 Bytes initialisiert mit 1, 3, 5 und 7
`Feld2   DB 5  DUP 19`	5 Bytes, jedes initialisiert mit 19
`Feld3   DD 10 DUP ?`	10 Double-Words (=40 Byte) ohne Initialisierung
`Feld4   DB 10 DUP '?'`	10 Bytes, jedes initialisiert mit 63 (=3Fh ='?')
`String1 DB 'Guten Tag!'`	10 Bytes, initialisiert mit dem Text Guten Tag!
`String2 DB 39,'Hallo',27h`	7 Bytes, initialisiert mit dem Text 'Hallo'
`String3 DB 50 DUP ' '`	50 Bytes, initialisiert mit Leerzeichen (=20h)

Beim Zugriff auf Variablen gibt es ein paar Dinge zu beachten. Die Größe von **Quelle** und **Ziel** muß übereinstimmen. Hierzu ein Beispiel, bezogen auf obenstehende Deklaration:

```
mov  AL,Zahl ; holt den Wert der Variablen Zahl (hier: 15) ins Byte-Register AL
```

Bei nachfolgendem Befehl gibt es eine Fehlermeldung beim Compilieren:

```
mov AX,Zahl
```

Das Register AX hat Word-Länge, die Variable nur Byte-Länge. Will man dagegen sozusagen „mit Gewalt“ eine Word-Variable laden (die 0Fh von Zahl als niederwertiges Byte und die 61h aus Zeichen als höherwertiges Byte ⇒ Ergebnis: 610Fh), dann lautet der Befehl:

```
mov  AX,W[Zahl]
```

Benötigt man dagegen die **Adresse** der Variablen Zahl im (Word-)Register AX, dann lautet der Befehl:

```
mov AX,Offset Zahl
```

8.3 Software-Interrupts

Es gibt keinen Befehl für das Ende eines Programs wie etwa „End.“ in PASCAL. Man kann es mit einem `ret` wie für das Ende eines Unterprogramms versuchen, und tatsächlich beendet sich das Programm mehr oder weniger zufällig. Die saubere Alternative ist es, das Betriebssystem - hier DOS - zu bitten, das Programm zu beenden. Für Betriebssystemfunktionen stehen eine ganze Reihe sogenannter Software-Interrupts[3] zur Verfügung. (**Achtung**: Die Interrupt-Nummern haben nichts mit den IRQ-Nummern zu tun!) Die Interrupt-Nummern 0 bis 1Fhex sind dem BIOS vorbehalten, ab 20hex

[3] Software-Interrupts sind Unterprogramme, die das Betriebssystem zur Verfügung stellt

gehören sie zum eigentlichen Betriebssystem.

Der Interrupt 21hex ist wohl der wichtigste unter DOS. Damit unter dem gleichen Interrupt vielfältige Funktionen abgearbeitet werden können, muß vor dem Aufruf des Interrupts noch die Funktionsnummer als Byte-Wert ins Register AH geladen werden. Je nach Funktionsnummer müssen noch weitere Werte in andere Register geschrieben werden. Die Funktionsnummer für ein Programmende lautet **4Chex**. Das Programmende sieht demnach also etwa so aus:

```
mov  AH,04C
int  021
```

Auch möglich ist folgendes: (Worin besteht der Unterschied?)

```
mov  AX,04C00
int  021
```

Eine weitere wichtige Funktionsnummer des Interrupt 21hex ist die Nummer 9 zur Ausgabe eines Textes. Dazu muss vor dem Aufruf das Register **DX** auf die Startadresse des Textes eingestellt werden. Der Text muss am Ende ein Dollarzeichen $ als Abschluß haben; das $-Zeichen wird nicht mit ausgegeben.

Weitere wichtige Funktionen des Interrupt 21hex sowie einiger wichtiger BIOS-Interrupts finden Sie auf der separaten Zusammenstellung.

9 Bedienung der Software

9.1 Der Assembler

Der Compiler für Assembler-Programme wird auch einfach **Assembler** genannt. Wir verwenden hier den **A86/A386** Shareware-Assembler, weil er besonders einfach in der Handhabung ist. Insbesonders bestimmte Direktriven, die andere Assembler im Kopf des Quellcodes erwarten, sind hier nicht notwendig. Zudem wird direkt eine ausführbare Datei im COM-Format erzeugt, ohne den bereits angesprochenen Umweg über eine Object-Datei.

A86 heißt die frei zugängliche Version mit der Beschränkung auf nur 8088-kompatible Strukturen, lizensierte User erhalten den A386 ohne diese Beschränkung. Die Handhabung beider Programme ist jedoch identisch. Ich beziehe mich hier immer auf den A386, da ich davon ausgehe, dass wir mehr als eine reine „Probierphase“ machen wollen, für die der A86 frei verfügbar ist.

Der A386 wird über die Kommandozeile gesteuert. Der Befehl lautet:

`A386` *Programmname*

Dabei steht *Programmname* für den Namen des Quelltextes **einschließlich** der für Assembler-Quelltexte üblichen Dateiendung `.ASM`. Soll das Programm `TEST.ASM` compiliert werden, lautet der Befehl also einfach:

```
A386 TEST.ASM
```

Durch diesen Befehl entsteht die ausführbare Datei `TEST.COM`.

Falls der Quelltext einen Fehler enthält, dann schreibt der A386 eine entsprechende Fehlermeldung **direkt unter die fehlerhafte Zeile in den Quelltext**. Die Originalversion wird dann mit der Endung `.OLD` gesichert. Es ist **nicht** erforderlich, nach Korrektur des Fehlers die Zeile mit der Fehlerbeschreibung von Hand zu löschen, das macht A386 automatisch beim nächsten Compilieren. **Achtung!** Damit diese Art der Fehlerbehandlung funktioniert, ist es **zwingend notwendig, nach dem letzten Befehl im Quelltext noch einen *Zeilenumbruch* einzufügen**, ansonsten entsteht aufgrund eines Bugs im A386 eine riesenlange „Mülldatei“. Die müsste im Falle eines Falles gelöscht werden, damit anschließend die Datei *Datei*`.OLD` in *Datei*`.ASM` zurück umbenannt werden kann.

9.2 Der Debugger

Wir verwenden hier den **D86/D386** Shareware-Assembler, der zum A86/A386 dazu gehört. Er wird mit dem Namen der **ausführbaren** Datei in der Kommandozeile aufgerufen. Das unter `TEST.ASM` erstellte Programm heißt als ausführbare Datei: `TEST.COM`
Der Aufruf im Debugger lautet also
`D386 TEST.COM`
oder etwas einfacher:
`D386 TEST`
aber **keinesfalls**:
`D386 TEST.ASM`
Letzteres ist ein häufig gemachter Fehler. Das Gemeine dabei ist, dass es in diesem Fall keine Fehlermeldung gibt, sondern es wird der **Quelltext** geladen, als ob er **ausführbaren Code** darstellen würde. Würde man diesen Code ablaufen lassen, wäre ein Absturz ziemlich sicher.

Wenn der Debugger D386 (beispielsweise mit einem „Hallo-Welt-Programm“) gestartet wird, ergibt sich etwa nachfolgendes Bild. Es werden ständig angezeigt:

- Der Quellcode des abzuarbeitenden Programms
- Die aktuellen Registerinhalte
- Die aktuelle Inhalte im Stack

```
# 0100 MOV AH,9                                D386 debugger, V4.05
  0102 MOV DX,TEXT                               Copyright 2000 Eric Isaacson
  0105 INT 021                                   All rights reserved.
  0107 MOV AH,04C                              For registered A86/D86 users only.
  0109 INT 021
TEXT:                                          Eric Isaacson           Visa/MC/Amex
  010B DEC AX                                  416 E. University Ave.
  010C POPA                                    Bloomington IN 47401-4739
  010D INSB                                       1-812-339-1811 voice,-335-1611 fax
  010E INSB                                       http://eji.com/a86/order
  010F OUTSW                                                A86+D86+A386+D386 is $80
  0110 AND B[BX+065],DL                        F10 key toggles windows
  0113 INSB                                    Alt-F10 key toggles HELP mode

EAX 0000_0000   i z e   1:
EBX 6C74_0000  IP 0100  2:
ECX 0000_00FF  CS 4653  3:
EDX 0000_4653  SS 4653  4:
ESI 0000_0100  DS 4653  5:
EDI 0000_FFFE  ES 4653  6:
EBP 0000_091C  FS 1CF3  7:
ESP 0000_FFFE  GS 0000  0:
```

Im Bereich unten links werden alle Register-Inhalte dargestellt. In der ersten Spalte sind es die Register mit Doppelwort-Länge von **EAX** bis **ESP**. Alle Inhalte werden grundsätzlich im Hex-Code angezeigt. Man kann erkennen, dass beispielsweise im **EDX**-Register der Wert **00004653** steht. Man kann dabei natürlich auch Teilregister ablesen, also **DX=4653**, **DL=53** oder **DH=46**.

In der zweiten Spalte rechts daneben kommt zunächst das **Flag-Register**. Die einzelnen Flags werden durch Buchstaben angezeigt, wenn sie gesetzt sind. Hier ist das Interruptflag (**i**), das Zero-Flag (**z**) und das Parity-Flag (**e**) gesetzt. Alle anderen Flags sind gelöscht. Darunter wird der Befehlszeiger **IP** angezeigt. Er zeigt immer auf die Offsetadresse des nächsten auszuführenden Befehles (hier: **0100**). Darunter befinden sich die Segment-Register **CS** bis **GS**. Da hier eine COM-Datei geladen wurde, enthalten **CS**, **SS**, **DS** und **ES** alle den gleichen Wert, nämlich **4653**.

Oben links werden die nächsten abzuarbeitenden Befehle angezeigt. Vor jedem Befehl steht die (Offset-) Adresse, an der er sich befindet. Vor dem ersten Befehl an der Adresse 0100 steht ein lila # auf schwarzem Grund. Das ist der Befehlscursor. Er zeigt immer auf den nächsten auszuführenden Befehl. Mit der Taste **F1** kann dieser Befehl ausgeführt werden. Der Befehlscursor springt dann eine Zeile tiefer und auch der Inhalt von IP erhöht sich entsprechend der Zahl der Bytes, die der Befehl belegt[4]. Es ist auch möglich, den Befehlscursor ohne Ausführen der Befehle durch den Debugger zu bewegen. Das geschieht mit den Cursortasten. Man kann ihn vor einen beliebigen Befehl stellen und diesen dann mit **F1** ausführen. Man kann auch bei Bedarf einen Befehl ausführen, der nicht im Programm steht. Man tippt ihn dann einfach von Hand ein und führt ihn mit <**Enter**> aus. Dieser Befehl wird in die letzte Zeile des roten Feldes geschrieben. Dort

[4] Die Befehle sind verschieden lang. Sie können zwischen 1 und 5 Byte belegen.

hin schreibt man auch die Kommandos für den Debugger[5].

Unten rechts ist noch ein freies Feld. Dort können (hinter den Ziffern 1 bis 7) Inhalte von Variablen im Speicher angezeigt werden. In der Zeile hinter der 0 werden die Daten auf dem Stack angezeigt. Im Beispiel liegt dort nichts, daher wird auch ein leerer Bereich angezeigt.

Unter dem roten Feld ist noch eine Zeile frei. Dort erfolgen Textausgaben, die ggf. das zu untersuchende Programm macht. Was wir zunächst noch wissen müssen, ist die Tastenkombination zur Beendigung des Debuggers. Es gibt zwei Möglichkeiten. Entweder gibt man „**q**“ ein und bestätigt es mit <**Enter**>, oder man drückt <**Alt-X**>. Weitere Infos findet man in der Dokumentation zum D386.

10 Das erste Programm

Der Konvention beim Erlernen von Programmiersprachen folgend ist unser erstes Programm ein Programm, das den Text „Hallo Welt!“ ausgibt. Im Gegensatz zu Hochsprachen gibt es unter Assembler **keinen** Befehl für eine Textausgabe. Wir können jedoch das Betriebssystem (hier DOS) bitten, das für uns durchzuführen. Dazu stehen uns allerlei Betriebssystem-Funktionen zur Verfügung, die mit einem Software-Interrupt aufgerufen werden. Die Nummer des Interrupt und auch die Funktionsnummer sind natürlich betriebssystem-spezifisch. DOS stellt hierfür den Interrupt 21hex zur Verfügung. (Unter Linux beispielsweise wäre es der Interrupt 80hex.) Innerhalb dieses Interrupt werden die unterschiedlichen Funktionen nach **Funktionsnummern** unterschieden. Bei DOS sucht das Betriebssystem diese Nummer im **AH-Register**. Wir müssen diese Nummer also in dieses Register hineinschreiben, **bevor** der Interrupt aufgerufen wird. Das ist ählich, wie bei einem Brief, auf den man die Empfängeranschrift schreiben muss, **bevor** man den Brief in den Briefkasten wirft. Informationen, welche Nummer was bedeutet und welche Werte sonst noch in welcher Richtung zwischen dem Assembler-Programm und dem Betriebssystem ausgetauscht werden müssen, findet man in der **Dokumentation des Betriebssystem-Herstellers**.

Die Funktionsnummer für eine Textausgabe ist die 9. Dabei erwartet DOS, dass der Text mit einem $ als „Textendemarkierung“ abgeschlossen ist. Weiterhin müssen wir DOS mitteilen, **wo** im RAM es den auszugebenden Text findet. DOS erwartet den Segment-Anteil der Adresse im Segment-Register **DS** (da sind sowieso unsere Variablen angelegt) und den Offset-Anteil der Adresse im Register **DX**. Wir müssen also von dem Interrupt-Aufruf auch noch DX richtig einstellen.

Hierzu ist es natürlich erforderlich, dass der auszugebende Text auch „irgendwo“ im Quelltext steht. Im Gegensatz zu Hochsprachen ist das immer ganz woanders, als beim

[5]Einzelheiten dazu bitte in der Dokumentation des A386 nachlesen.

Befehl zur Ausgabe dieses Textes. Dabei ist darauf zu achten, dass der Prozessor nicht versehentlich den Ausgabetext als auszuführende Befehle angeboten bekommt, denn das führt eigentlich immer zu einem Absturz. Man sollte alle Variablen – also auch Ausgabetexte – in einem mehr oder weniger getrennten Bereich Bereich anlegen, wie im Kapitel *Grundstruktur von Assemblerprogrammen*dargestellt. Eine Verpflichtung dazu gibt es freilich nicht. Im nachfolgenden Beispielprogramm ist der Text **hinter** dem Programmende angelegt.

Beispiel für ein Hallo-Welt-Programm:

```
mov  DX,Offset Text      ; Lade die Adresse des Textes ins Register DX
mov  AH,09               ; setze AH auf Funktionsnummer zur Textausgabe
int  021                 ; führe die Textausgabe durch
mov  AH,04C              ; setze Funktionsnummer für Programmende
int  021                 ; beende das Programm
Text DB 'Hallo Welt!$'   ; hier steht der Text zur Ausgabe, mit $ abgeschlossen
```

Aufgabe 1:

1. Geben Sie das Programm mit dem Namen `HALLO.ASM` ein, compilieren Sie es und testen Sie es, indem Sie es an der Kommandozeile aufrufen.
2. Laden Sie das Programm in den Debugger D386 und führen Sie es im Einzelschrittmodus aus. Beobachten Sie dabei die Veränderungen der Registerwerte.
3. Schreiben Sie das Programm so um, dass der auszugebende Text möglichst weit am Anfang des Quelltextes steht (siehe Kapitel *Grundstruktur von Assemblerprogrammen*); compilieren und testen Sie es, wie beim Ursprungsprogramm. Vergleichen Sie die resultierenden Längen der COM-Dateien.

11 Sprungbefehle

11.1 Allgemeines zu Sprungbefehlen

Sie haben im Kapitel *Grundstruktur von Assemblerprogrammen*schon den Befehl `JMP` zum Überspringen der Variablendeklaration kennen gelernt. Als Sprungziel verwendet man normalerweise Sprungmarken, sogenannte Labels, einen Namen mit Doppelpunkt dahinter (Beispiel `START:`). Die Groß-/Kleinschreibung spielt keine Rolle, jedoch dürfen nur Buchstaben (ohne Umlaute), Unterstriche und mit gewisser Einschränkung Ziffern verwendet werden.

Es gibt **globale** und **lokale** Labels. **Globale Labels** dürfen in einem Programm nur ein einziges Mal vorkommen. Man verwendet sie für Unterprogrammadressen und wesentliche Abschnitte in einem Programm. Will (oder muss) man mit Schleifen und kurzen

Sprüngen arbeiten, dann gehen einem schnell die sinnvollen Namen aus. Hier helfen lokale Labels weiter. **Lokale Labels** dürfen beliebig oft verwendet werden. Damit der Compiler weiß, welche gemeint ist, muss daher angegeben werden, ob vorwärts oder rückwärts gesprungen werden soll. Er nimmt dann die erste Marke in der angegebenen Richtung mit diesem Namen.

Lokale Labels beginnen mit **genau einem Buchstaben**, danach folgen **Ziffern**.
Beispiele: **L1:** **S20:** **P12345:** **K045:**

Bei Sprüngen im Programm gibt es **drei unterschiedliche** Sprungweiten: `FAR`, `NEAR` und `SHORT`, entsprechend den Adressbreiten **Doubleword**, **Word** und **Byte**. Erstere führen einen Sprung in ein anderes Speichersegment aus; Einzelheiten werden später besprochen. Wie bekannt, haben alle Adressen innerhalb eines Segmentes Word-Breite. Dem entspricht der normale (`NEAR`) Sprung. Für einen sehr kurzen (`SHORT`) Sprung um maximal 127 Bytes vorwärts oder rückwärts im Programm genügt auf Maschinensprachenebene ein einziges Byte zur **relativen** Adressierung; folglich wird das Programm kürzer. Wichtiger aber ist: alle **bedingten** Verzweigungen, die anschließend besprochen werden, können nur als `SHORT` codiert werden. Für einen Sprung zu einer lokalen Variablen unterstellt der Compiler A386 ebenfalls einen `SHORT JMP`. Hieraus resultieren oft Fehlermeldungen beim Compilieren, der Sprung sei zu groß.

Bei einem Sprung zu einer **lokalen** Sprungmarke unterstellt A386 zunächst immer einen Sprung **rückwärts**. Soll **vorwärts** gesprungen werden, muss dies mit einemGrößer-Zeichen explizit angegeben werden. Beispiele:

```
jmp  S5    ; SHORT JMP rückwärts zur letzten lokalen Sprungmarke S5
jmp  >S5   ; SHORT JMP vorwärts zur nächsten lokalen Sprungmarke S5

jmp  ST5   ; NEAR JMP zur globalen Sprungmarke ST5, egal, wo diese ist.
           ; Aber Achtung: Ist ST5 als Word- oder Doubleword-Variable deklariert
           ; worden, dann springt das Programm zu der Stelle, die der Wert der
           ; Variablen angibt: ein NEAR JMP bei Word-Variable und ein FAR JMP
           ; bei Doubleword-Variable.
```

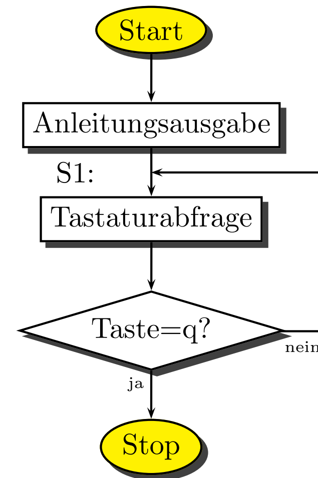
Soll ausdrücklich ein `SHORT JMP` zu einer **globalen** Sprungmarke ausgeführt werden, so kann das mit angegeben werden:

```
jmp short ST5
```

Das kann sinnvoll sein, wenn das Programm auf minimale Größe und maximale Ablaufgeschwindigkeit optimiert werden soll.

11.2 Bedingte Sprünge, Verzweigungen, Schleifen

Es gibt eigentlich kein Programm, in dem nicht Verzweigungen vorkommen. In Hochsprachen ist die Bedingung und die Verzweigung immer **Bestandteil des gleichen Befehls**; nicht so bei Assembler. Hier geht es immer **zweistufig**: Zunächst wird eine Operation durchgeführt oder eine Bedingung abgefragt, wodurch bestimmte Flags gesetzt werden, z. B. `cmp AL,'q'`. Mit diesem Befehl wird der Inhalt des Registers **AL** mit dem Wert des ASCII-Zeichens **q** (=71hex=113dez) verglichen. Alle möglichen Vergleiche werden gleichzeitig durchgeführt, beispielsweise auf größer, kleiner oder gleich. Die entsprechenden Flags werden gesetzt oder gelöscht. Erst im **nachfolgenden Befehl** erfolgt ein **bedingter Sprung**, ein Sprung, der nur ausgeführt wird, wenn ein bestimmtes Flag gesetzt oder gelöscht ist, wie etwa: `JNE S1`. JNE steht für „Jump not even“, also für „Springe zum Label S1, wenn nicht gleich“. Der Sprung wird demnach nur dann ausgeführt, wenn das Z-Flag gelöscht ist.



Aufbau einer Schleife

Zu nachfolgendem Beispiel gehört obenstehendes **Flussdiagramm**. Man erkennt, dass zunächst ein Anleitungstext ausgegeben wird. Danach wird die Tastatur abgefragt. Wenn die Taste **q** gedrückt wurde, beendet sich das Programm. Anderenfalls durchläuft das Programm die Schleife, die an der Stelle `S1:` beginnt. Die Tastatur wird erneut abgefragt, usw., so lange, bis die Taste **q** betätigt wurde.

```
; Programm ABFRAGE
  mov  DX,TEXT; Adresse des Anleitungstextes
  mov  AH,09   ; Funktionsnummer für Textausgabe
  int  021     ; Textausgabe durchführen
S1:            ; Schleifenanfang
  mov  AH,010  ; Funktionsnummer zum Lesen eines Eingabezeichens
  int  016     ; Tastatur-Interrupt - er liefert das Zeichen in AL
  cmp  AL,'q'  ; ist das Zeichen ein "q"?
  jne  S1      ; wenn nein, Sprung zurück nach S1 (Schleife)
  mov  AH,04C  ; Funktionsnummer für Programmende
  int  021     ; Programmende durchführen
;------ Variablenbereich: ------------
TEXT:  db 'Programmende mit Taste q!$'
```

Ein Hinweis zur Funktion des Programms: Mit dem **Interrupt 16h** wird die Tastatur abgefragt. Die **Funktion 10h** dieses Interruptes wartet, bis eine Taste gedrückt wird. Das Zeichen wird dann in **AL** zurückgegeben. (Bei Sondertasten – Funktionstas-

ten, Cursortasten usw, –wird in AL eine 0 oder der Wert E0h geliefert, der eigentliche Zeichencode steht dann in **AH**.)

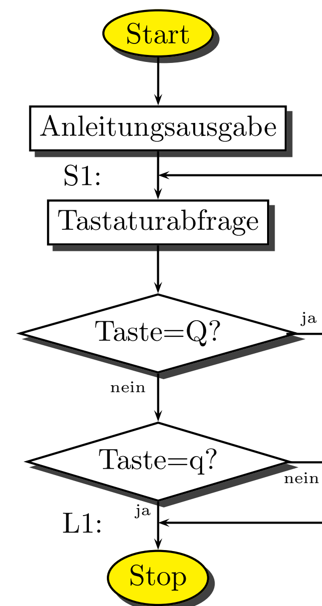
12 Übungsaufgaben, Teil 1:

12.1 Aufgabe 1.1

Tippen Sie das Programm `ABFRAGE.ASM` ab,[6] compilieren Sie das Programm und testen Sie es aus. Laden Sie es in den Debugger D386 und testen Sie es im Einzelschrittmodus. ***Frage:*** Warum steht in der ersten Programmzeile (Zeile 7 im Quelltext) kein `Offset` vor `TEXT`?

12.2 Aufgabe 1.2

Ändern Sie das Programm so, dass es nicht nur beim kleinen **q**, sondern auch beim großen **Q** abbricht! (Neuer Name: `ABFRAGE1.ASM`) Beachten Sie, dass eine **gleichzeitige** Abfrage auf **q** und **Q** in einem einzigen Befehl wie in einer Hochsprache **nicht** möglich ist. Orientieren Sie sich dazu an nebenstehendem Flussdiagramm!



ABFRAGE1.ASM

Beachten Sie beim Schreiben dieses Programms – und auch aller weiterer – bitte folgende Konventionen:

- Es hat sich als sehr sinnvoll erwiesen, den Quellcode **strukturiert** zu schreiben. Das bedeutet, dass alle Sprungmarken am Anfang einer Zeile stehen, die Befehle hingegen etwas eingerückt geschrieben werden. (In dieser Form ist auch das Beispielprogramm `ABFRAGE.ASM` geschrieben.)
- Gewöhnen Sie sich bitte **von Anfang an** daran, **sinnvolle Kommentare** in den Quelltext zu schreiben. Bei Assembler ist das noch wichtiger, als schon bei Hochsprachen! Dazu können auch strukturierende Striche gehören, wie im Beispielprogramm zur Abtrennung des Variablenbereiches gemacht.

Fertigen Sie immer ein **Flussdiagramm** an, und zwar **bevor** Sie auch nur ein einziges Zeichen Quelltext geschrieben haben. Sofern Sprungmarken erforderlich sind, tragen Sie deren Namen mit im Flussdiagramm ein! Nur so wird

[6]Meine Schüler im BKT-Assembler-Kurs finden diese Datei auch im Austauschverzeichniss des Schulservers

es gelingen, einen Überblick über das Programm zu behalten. Oben ist ein mögliches Flussdiagramm zum Programm `ABFRAGE1` dargestellt. Mit `S1:` und `L1:` sind im Flussdiagramm die Namen der Sprungmarken (Labels) eingetragen.

12.3 Aufgabe 1.3

Ergänzen Sie das Programm `ABFRAGE1.ASM` so, dass jedes gedrückte Zeichen zuerst auf dem Bildschirm dargestellt wird, bevor geprüft wird, ob es ein **q** oder **Q** ist! Verwenden Sie hierzu die **Funktion 0Eh des Interrupt 10h**! (Neuer Name: `ABFRAGE2.ASM`) Tragen Sie die Ergänzung auch in das zugehörige Flussdiagramm ein!

12.4 Aufgabe 1.4

Entwerfen Sie ein Programm mit dem Namen `10TASTEN.ASM`, das sich nach genau 10 Tastendrücken beendet. Ansonsten gibt das Programm bein Start nur eine kurze „Anleitung“ aus.

Entwerfen Sie dazu zunächst ein passendes Flussdiagramm. Ein Tipp: Lesen Sie sich einmal genau die Beschreibung zum Befehl `LOOP` durch!

12.5 Aufgabe 1.5

Entwerfen Sie ein Programm mit dem Namen `ZIFFER.ASM` mit folgenden Eigenschaften: Nach dem Start schreibt das Programm eine kurze „Bedienungsanleitung“ auf den Schirm und wartet dann auf eine Tastatureingabe. (Hinweis dazu: die Codes für einen Zeilenumbruch sind `0D,0A`.) Wird eine **1** gedrückt, erscheint der Text:

```
Dies ist die Taste 1.
```

Bei der 2, 3 oder 4 erscheint ein entsprechender Text mit dem jeweiligen Zeichen. Nur bei Zifferntasten von 0 bis 9 soll eine Ausgabe erfolgen, alle anderen Zeichen werden ignoriert. Mit der Taste <**Esc**> wird das Programm beendet. Hinweis dazu: Stellen Sie mit Hilfe des letzten Programms und des Debuggers **D386** fest, welchen Code die Taste <**Esc**> bzw. die jeweilige Zifferntaste erzeugt.

13 Stapelverarbeitung

Zu jedem Programm gehört ein Stapel (oder Stack). Dort können bei Bedarf Werte abgelegt und wieder zurückgeholt werden. Beim Aufruf einer Unterfunktion wird hier die Rücksprungadresse abgelegt. In höheren Programmiersprachen werden dort auch lokale Variable angelegt, die nur innerhalb einer Unterfunktion gültig sind. Bei der Arbeit mit dem Stapelspeicher ist peinlich darauf zu achten, was in welcher Reihenfolge dort abgelegt wird, damit es in genau der umgekehrten Reihenfolge wieder zurückgenommen wird. Ansonsten ist ein Programmabsturz unausweichlich. Ausnahme: Sie wollen

bewusst einen Wert über den Stapel in ein anderes Register übertragen.

Zur Verwaltung dient das Prozessorregister **SP** (bzw. **ESP** im Protected Mode), der Stack-Pointer. Er zeigt immer auf die Adresse, an der eine Word-Wert oder ein Doppelwort abgelegt ist. Byte-Werte können nicht einzeln abgelegt werden, notfalls kommt ein zweites (überflüssiges) Byte mit auf den Stapel. Dies geschieht mit dem Befehl `PUSH`. Dieser Befehl bewirkt folgendes: Zunächst vermindert der **SP** seinen Wert um 2, dann wird der zu übertragene Wert an die Stelle gespeichert, auf die SP zeigt. Der Stapel wächst also gewissermaßen nach unten. Bei einem COM-Programm ist der Startwert für SP immer 0FFFE, er zeigt also auf die beiden letzten Bytes in dem Segment. Mit dem Befehl `POP` wird zuerst das Wort, auf das er jetzt zeigt, zurückgegeben und anschließend SP um 2 erhöht. Der Zustand vor dem letzten `PUSH` ist also wieder erreicht.

Beispiele:

```
push AX         ; Inhalt des Registers AX auf den Stapel legen
pushf           ; Inhalt des Flag-Registers auf den Stapel legen
push BP         ; Inhalt des Basis-Zeigers auf den Stapel legen
push Variable   ; Inhalt einer Word-Variable auf den Stapel legen
pop  Variable   ; letzten Wert vom Stapel holen und in Word-Variable speichern
pop  BP         ; letzten Wert vom Stapel holen und in den den Basis-Zeiger speichern
popf            ; letzten Wert vom Stapel holen und in das Flag-Register speichern
pop  AX         ; letzten Wert vom Stapel holen und in das Register AX speichern
```

Aufgepaßt! Was passiert bei folgender Sequenz?

```
push  AX
popf
```

Durch die vorangehende Sequenz gelangt der Inhalt des Registers **AX** ins **Flag-Register**! Ist beispielsweise in AX das Bit 9 Null, dann sind ab sofort alle Hardware-Interrupts gesperrt, so zum Beispiel der Tastatur-Interrupt, der bei jedem Tastendruck ausgelöst wird. Der Rechner „hängt“ sich auf, reagiert nicht mehr auf die Tastatur.

Auch bei einem Aufruf eines Unterprogramms mit `CALL` wird der Stapel benutzt. Die Adresse, zu der am Ende des Unterprogramms mit `RET` zurückgesprungen werden muss, wird auf den Stapel gelegt. Extrem wichtig ist daher, dass die durchgeführten `PUSH`-Befehle durch eine genau gleiche Anzahl `POP`-Befehle wieder aufgehoben werden. Bleibt ein Wert auf dem Stapel liegen, oder wird einer zuviel entfernt, ist ein Absturz nahezu unvermeidlich, denn beim nächsten Rücksprung aus einem Unterprogramm wird ein falscher Wert als Rücksprungadresse interpretiert. Aus diesem Grund ist es **dringend** angeraten sofort auch einen `POP`-Befehl zu schreiben, nachdem man einen `PUSH`-Befehl geschrieben hat. Notwendige andere Befehle fügt man dann erst anschließend dazwischen ein.

14 Übungsaufgaben, Teil 2

14.1 Aufgabe 2.1

Betrachten Sie nachfolgendes Programm `ZEICHEN.ASM`.[7] Tippen Sie es ab, compilieren Sie es und testen Sie es. Die gewünschte Funktion geht aus den Kommentarzeilen hervor. Warum läuft es nicht ordnungsgemäß? Testen Sie das Programm mit dem **D386** im Einzelschrittmodus aus. Beheben Sie den Fehler, wenn Sie ihn lokalisiert haben.

```
; Programm ZEICHEN.ASM
jmp  START; Sprung über den Bereich der Variablen
;---- Ab hier Deklaration von Variablen ----
TEXT1: db 'Bitte eine Taste drücken, Abbruch mit <Esc>',0A,0D,'$'
TEXT2: db 0A,0D,'Sie haben die Taste '
BUCHSTABE db 'x'
  db ' gedrückt.$' ; (Resttext von TEXT2)
;---- Ende der Variablendeklaration ------
START:
  mov DX,TEXT1 ; Adresse des Anleitungstextes nach DX
  mov AH,09    ; Funktionsnummer für Stringausgabe
  int 021      ; Ausgabe des Anleitungstextes Text1
S1:
;-- Ein Zeichen wird von der Tastatur geholt:
  mov AH,010
  int 016
;--
  mov BUCHSTABE,AL ; Das Zeichen wird in den Ausgabetext "eingeflickt"
;-- Es folgt die Ausgabe des Textes:
  mov DX,TEXT2
  mov AH,09
  int 021
;--
  cmp AL,01B   ; war das Zeichen das <Esc>?
  jne S1       ; wenn nein, weiter in Schleife
  mov AH,04C   ; sonst: Programmende einleiten
  int 021
```

[7] Meine Schüler im BKT-Assembler-Kurs finden diese Datei auch im Austauschverzeichniss des Schulservers

14.2 Aufgabe 2.2

Erstellen Sie ein Programm mit dem Namen TASTEN.ASM mit folgenden Eigenschaften:

- Ein Zeichen wird von der Tastatur geholt und auf dem Bildschirm dargestellt.
- Solange nicht die Taste <**F1**> gedrückt wird, wird ein Zeichen nach dem anderen auf dem Bildschirm dargestellt.
- Beim Druck auf eine beliebige Funktionstaste erscheint am Anfang einer **neuen Zeile** ein kurzer Text:
  `Sie haben eine Funktionstaste gedrückt.`
- Wenn die Taste <**F1**> gedrückt wird, beendet sich das Programm.
- Versehen Sie das Programm ausreichend mit Kommentaren!

Ein Tip dazu: Sie können mit Hilfe des Debuggers D386 herausfinden, welche Tastencodes die Funktionstasten erzeugen. **Erstellen Sie *vor* dem Schreiben des Programms ein Flussdiagramm!** Bemühen Sie sich, dass Flussdiagramm nicht in die **Breite** wachsen zu lassen. Eine einspaltige gestreckte Form entspricht besser dem zu erstellenden Quellcode.

15 Unterprogramme

Ein Unterprogramm beginnt sinnvollerweise (aber nicht notwendigerweise) mit einem globalen Label. Es endet mit einem `RET`. Aufgerufen wird es mit `CALL`. Der `CALL`-Befehl bewirkt folgendes:

- Die Adresse des nächsten auszuführenden Befehles wird als Rücksprungadresse auf den Stapel gelegt.
- Danach erfolgt ein Sprung an die Startadresse des Unterprogramms.

Nachdem das Unterprogramm abgelaufen ist, trifft der Prozessor auf den `RET`-Befehl. Dadurch wird der Prozessor veranlasst, die Rücksprungadresse vom Stapel zu nehmen (hoffentlich liegt sie da noch ...) und den Befehlszeiger auf diese Adresse einzustellen, um dort weiter zu machen.

Nachfolgendes Beispiel holt ein Zeichen von der Tastatur und gibt es aus. Wurde <**Esc**> gedrückt, beendet sich das Programm. Anderenfalls wird das nächste Zeichen von der Tastatur abgefragt.

```
jmp  START        ; Sprung zum Start des Hauptprogramms
;-------
HOLE_TASTE:       ; Unterprogramm zur Abfrage der Tastatur
                  ; Rückgabe des Tastencodes in AX
  mov  AH,010     ; Funktionsnummer für Tastaturinterrupt
  int  016        ; ausführen des Tastaturinterrupt
  ret             ; Rücksprung ins Hauptprogramm
;-------
SCHREIBE_BUCHSTABEN: ; Unterprogramm Bildschirmausgabe
                  ; Übergabe des zu schreibende Buchstabens in AL
  mov  AH,0E      ; Funktionsnummer Zeichenausgabe
  int  010        ; Video-Interrupt (Zeichenausgabe durchführen)
  ret             ; Rücksprung ins Hauptprogramm
;-------
START:            ; Hier beginnt das Hauptprogramm
  call HOLE_TASTE ; Aufruf Unterprogramm
  push AX         ; Der Inhalt von AX (enthält den Tastencode) sichern
  call SCHREIBE_BUCHSTABEN ; Aufruf Unterprogramm
  pop  AX         ; Den Tastencode vom Stapel nach AX zurückholen
  cmp  AL,01B     ; Wurde Taste <Esc> gedrückt?
  jne  START      ; wenn nein, weiter in Schleife
  mov  AX,04C00 ; sonst Programmende vorbereiten
  int  021        ; Programmende durchführen
```

An welcher Stelle im Quellcode die Unterprogramme liegen, ist völlig gleichgültig. Es ist jedoch zweckmäßig, sie an den Anfang zu legen, wie es ja auch in den meisten Hoch-

sprachen gemacht wird, also auch in diesem Beispiel.

Weiterhin ist es äußerst zweckmäßig, **aus einem Unterprogramm heraus *keine* direkten Zugriffe auf Variablen** zu machen. Sie werden dadurch vielseitiger einsetzbar, übersichtlicher und weniger fehleranfällig. Statt dessen übergibt man dem Unterprogramm besser notwendige Parameter in **Registern**[8] oder auf dem **Stapel**[9]. Werte, die das Unterprogramm an den aufrufenden Programmteil zurückliefern soll, werden üblicherweise je nach Größe in **AL**, **AX** oder **EAX** zurückgegeben. So machen es alle Hochsprachen; deshalb sollte man es nicht ohne Not unter Assembler anders machen, auch wenn das natürlich möglich ist.

Ganz wichtig ist auch eine entsprechende Dokumentation. **In den Kopf eines jeden Unterprogramms gehört *immer* ein Kommentar über die Übergabe-Parameter**, wie dies auch im Beispiel gemacht wurde.

Es gibt noch einen Punkt, den man beim Arbeiten mit Unterprogrammen **unbedingt** beachten muss. In jedem Programmabschnitt werden irgendwelche Register benutzt. Das gilt sowohl für ein Unterprogramm als auch für den aufrufenden Prorgammteil. Möglicherweise erwartet der aufrufende Programmteil, dass irgendwelche Daten in den Registern erhalten bleiben. Wenn aber das Unterprogramm eines dieser Register selbst verwendet, dann ist das nicht mehr gegeben. Dagegen ist zweierlei Abhilfe möglich:

1. Vor dem Aufruf eines Unterprogramms werden die fraglichen Register auf dem Stapel gesichert.

2. Das Unterprogramm sichert selbst alle Register, bevor es diese nutzt und holt die Inhalte wieder zurück, bevor es sich beendet.

Beide Verfahren sind möglich und auch sinnvoll. Beide haben aber sowohl Vorteile, als auch Nachteile.

Im ersten Fall muss man bei **jedem** Aufruf eines Unterprogramms diese Register sichern. Man muss auch immer genau wissen, welche Register das fragliche Unterprogramm verändert. Verwendet man Unterprogramme aus Sammlungen (Bibliotheken), dann sind diese Informationen nicht immer sofort klar erkennbar. Gewöhnt man sich statt dessen an, alle Unterprogramme so zu schreiben, dass nach seiner Beendigung alle Register wiederhergestellt sind, dann werden eventuell Register gesichert und wiederhergestellt, die das aufrufende Programm gar nicht benötigt. Die Unterprogramme werden also eventuell unnötig verlangsamt.

[8]Die Übergabe in Registern kennen wir bereits von Hardware-Interrupts, die auch eine Form von Unterprogramm darstellen.

[9]Die Übergabe auf dem Stapel – sie wird von vielen Hochsprachen bevorzugt – wird später im Kurs erläutert.

Welche Strategie man verwendet, muss jeder für sich selbst entscheiden. Wer ganz auf Nummer Sicher gehen will, kann natürlich einerseits die Unterprogramme so schreiben, dass alle Register erhalten bleiben und sicherheitshalber trotzdem vor dem Aufruf kritische Register noch einmal sichern.

Es ist auf jeden Fall empfehlenswert, im Kopf eines Unterprogramms im Kommentar immer anzugeben, ob und wenn ja, welche Register verändert werden. Dann hat man relativ einfach einen Überblick.

16 Übungsaufgaben, Teil 3:

16.1 Aufgabe 3.1

Schreiben Sie ein Programm mit dem Namen `ZIFFERN.ASM` mit folgende Bedingungen:

- Beim Programmstart erscheint der Text:
  `Bitte Ziffern eingeben, Programmende mit <Esc>`
- Wird eine Ziffer eingegeben, dann erscheint diese auf dem Bildschirm.
- Wird die Taste <**Enter**> gedrückt (Code 0D), springt der Cursor an den nächsten Zeilenanfang. Dies wird erreicht, indem die die beiden Zeichen 0D (Cursorsprung an Zeilenanfang) und 0A (Cursorsprung in nächste Zeile) nacheinander ausgegeben werden.
- Wird die Taste <**ESC**> gedrückt (Code 01B), beendet sich das Programm.
- Wird eine andere Taste gedrückt, wird ein Warnton ausgegeben. (Dies geschieht durch Ausgabe des Zeichens 07.)
- Verwenden Sie Unterprogramme, wo dies sinnvoll ist.
- Kommentieren Sie das Programm. Geben Sie ggf. an, welche Übergabewerte ein Unterprogramm erwartet oder an den aufrufenden Programmteil zurückgibt.

Bevor Sie mit dem Schreiben des Programms beginnen, fertigen Sie ein Flussdiagramm an! Achten Sie dabei darauf, dass Sie nur Verzweigungsfragen eintragen, die es unter Assembler auch gibt. Beispielsweise lässt sich die Frage: *Ziffern? ja – nein* durch keine Assemblerbefehle abbilden. Sie können lediglich fragen, ob der Code eines Zeichens **größer**, **kleiner** oder **gleich** einem Vergleichswert ist. Es ist sinnvoll, **separate** Flussdiagramme für das Hauptprogramm und diverse Unterprogramme zu erstellen, falls diese notwendig sind. Das macht alles übersichtlicher.

Hier ein paar Hinweise zu einigen möglicherweise verwendbaren Sprungbefehlen:

`JA`	Sprung wenn vorzeichenlos größer (**J**ump if **A**bove)
`JAE`	Sprung wenn vorzeichenlos größer oder gleich (**J**ump if **A**bove or **E**ven)
`JB`	Sprung wenn vorzeichenlos kleiner (**J**ump if **B**elow)
`JBE`	Sprung wenn vorzeichenlos kleiner oder gleich (**J**ump if **B**elow or **E**ven)
`JE`	Sprung wenn gleich (**J**ump if **E**ven)
`JNE`	Sprung wenn ungleich (**J**ump if **N**ot **E**ven)

16.2 Aufgabe 3.2

Schreiben Sie ein Programm mit dem Namen `CURSOR.ASM` mit folgenden Eigenschaften:

- Wird eine Cursortaste betätigt, dann bewegt sich der Cursor über den Bildschirm um einen Schritt in die jeweilige Richtung. Würde der Cursor dadurch den Bildschirmbereich verlassen, bewegt er sich nicht, statt dessen ertönt ein Warnton.
- Die Taste <**Esc**> beendet das Programm. Zuvor springt der Cursor an die Stelle, wo er beim Programmstart war.

Hinweis 1: Finden Sie die Tastencodes der Cursortasten in **AX** bei Verwendung der **Funktion 10h** des **Interrupt 16h** mit Hilfe des Debuggers D386 heraus.

Hinweis 2: Suchen Sie geeignete Funktionen zur Cursor-Bewegung beim Video-Interrupt bei den BIOS-Interrupts.

Verwenden Sie Unterprogramme für die Ausführung der jeweiligen Aktion. Fertigen Sie – wie immer – vor der Programmerstellung Flussdiagramme zum Hauptprogramm und zu einzelnen wichtigen Unterprogrammen an!

16.3 Aufgabe 3.3

Entwerfen Sie ein Programm mit dem Namen `HTASTEN.ASM`, mit dem der Zeichencode eines beliebigen Zeichens auf dem Bildschirm als Hexadezimalzahl dargestellt wird. Nach jedem dargestellten Zeichen springt der Cursor an den Anfang der nächsten Zeile. Das Programm wird mit der Taste <**Esc**> beendet, nachdem es noch dargestellt wird.

Hier ein paar Hinweise für einige Befehle, die möglicherweise benötigt werden:

```
and  AL,0F   ; löscht die oberen 4 Bit im Register AL
and  AL,0F0  ; löscht die unteren 4 Bit im Register AL
shr  AL,4    ; verschiebt die oberen 4 Bits in AL nach unten
xchg AL,AH   ; tauscht die Registerinhalte von AL und AH
add  AL,'0'  ; wandelt Zahlenwert in Ziffer, falls zwischen 0 und 9, konkret:
             ; zum Wert in AL wird der Wert des ASCII-Zeichens 0 hinzugezählt
```

Der Aufbau des Programmes soll folgendermaßen aussehen:

- Zuerst gibt das Programm eine kurze Bedienungsanleitung aus.
- Anschließend wird ein Zeichen von der Tastatur geholt.
- Das Zeichen wird umgewandelt in **zwei** ASCII-Zeichen, die den Hex-Code des Zeichens darstellen. Diese Umwandlung soll in einem Unterprogramm mit dem Namen `HEXCODE` vorgenommen, für das folgende Bedingungen gelten soll: Es erwartet das **umzuwandelnde Zeichen** im Register **AL**. Es liefert die beiden ASCII-Zeichen in **AL** und **AH** zurück, wobei der höherwertige Teil in AH stehen soll. Nach der Rückkehr aus dem Unterprogramm sollen alle Register außer AX erhalten bleiben.

- Die beiden ASCII-Zeichen werden in einen Ausgabetext eingesetzt, der anschließend auf dem Bildschirm ausgegeben wird.
- **Nach** der Textausgabe wird das (hoffentlich rechtzeitig gesicherte) Zeichen geprüft, ob es sich um das Zeichen <**Esc**> handelte. In diesem Fall beendet sich das Programm, anderenfalls springt es zurück an die Stelle nach der Ausgabe der Bedienungsanleitung.

Die Ausgabe des Programms sieht nach Eingabe der Zeichen `a m ß` wie folgt aus:

```
Bitte eine Taste drücken!
Beenden mit <Esc>
Sie haben die Taste a gedrückt, Zeichencode 61 hex
Sie haben die Taste m gedrückt, Zeichencode 6D hex
Sie haben die Taste ß gedrückt, Zeichencode E1 hex
```

Wenn das Programm ordnungsgemäß läuft, ergänzen Sie es dahingehend, dass nicht nur der Hexcode von **AL** sondern auch der Hexcode von **AH** auf dem Bildschirm dargestellt wird, denn beide zusammen ermöglichen erst eine korrekte Erkennung, welche Taste gedrückt wurde. Konkret: Das BIOS liefert bei „normalen“ Tasten den Zeichencode des ASCII-Zeichens in **AL** und den Code, den die Tastatur erzeugt, in **AH**. Dadurch ist es beispielsweise möglich, das Pluszeichen im Ziffernblock vom Pluszeichen aus dem linken Tastenfeld zu unterscheiden. Auch beide Enter-Tasten erzeugen unterschiedlichen Code. Wird keine „normale“ Taste sondern eine Sondertaste (Funktionstaste, Cursortaste o. ä.) gedrückt, dann steht der eigentliche Code nicht in **AL** sondern in **AH**. Um dies zu signalisieren ist in diesem Fall AL= 00h oder E0h. Das ergänzte Programm soll `HTASTEN2.ASM` heißen.

17 Zeiger und Segment-Override

17.1 Zeiger

Was ist ein Zeiger? Ein Zeiger enthält **nicht den Variablenwert**, sondern **die Adresse der Variablen**. Dazu ein praktisches Beispiel aus dem täglichen Leben, das die Funktion eines Zeigers veranschaulichen soll.

Ich möchte einem Kollegen eine Datei per Email schicken. Dazu habe ich zwei Möglichkeiten:

1. Ich füge die Datei als Anhang an die Email an.
2. Ich stelle die Datei irgendwo ins Netz und schicke meinem Kollegen den zugehörigen Link.

Im ersten Fall habe ich direkt die Datei weitergegeben. Je nach Dateigröße hat das lange gedauert. Im zweiten Fall habe ich nur einen **Zeiger** auf die Datei verschickt. Das ging flott, da ich nur die vergleichsweise kurze Adresse übermitteln musste. Der Kollege weiß dann aber, wo er die Datei findet.

Sinngemäß genau so funktionieren Zeiger beim Programmieren. Ich kann direkt einen **Variablenwert** übertragen, ich kann aber alternativ auch eine **Adresse** einer Variablen übertragen. **Der Inhalt einer Zeigervariablen ist also stets die *Adresse* einer tatsächlichen Variablen.** Wir kennen das auch schon von der Funktion 09 des Interrupt 21h[10] in DOS. Da man nicht den ganzen String mit allen Zeichen in ein Prozessorregister packen kann, wird hier das Register **DX** als Zeiger verwendet. Die DOS-Funktion „weiß“ damit, wo sie den String finden kann, den sie ausgeben soll.

Soll ein (Assembler-)Programm irgendwelche Aktionen mit vielen Elementen eines Arrays[11] durchführen, ist es nicht machbar, den immer wieder gleichen Befehl für jedes Element des Arrays mit jeder Adresse einzeln zu schreiben. Statt dessen verwendet man eine Schleife, in der ein **Zeiger** immer wieder weiter gestellt wird, der dann zur Adressierung verwendet wird. Der Befehl, mit dem diese Aktion durchgeführt wird, muss dann nur **einmal** mit dem Zeiger im Programm stehen.

Die Register **SI**[12], **DI**[13] und **BX** können als Zeiger verwendet werden. Auch Register **BP**[14] wird als Zeiger verwendet, jedoch kann BP (normalerweise) nur für Stack-Operationen verwendet werden.

[10] Ausgabe eines mit $ abgeschlossenen Strings
[11] auch ein String ist ein Array
[12] SI steht für **source-index**, auf Deutsch: Quellzeiger
[13] DI steht für **destination-index**, auf Deutsch: Zielzeiger
[14] BP steht für **base-pointer**, auf Deutsch: Basiszeiger

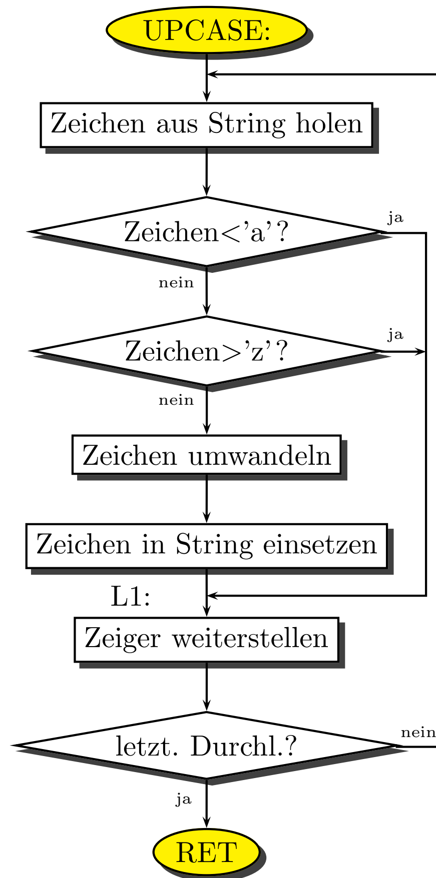
Hier die Befehlsstruktur anhand von Beispielen:

```
mov  AX,SI          ; Dies hat mit Zeigern nichts zu tun. Der Inhalt von SI wird
                    ; nach AX geladen.
mov AX,[SI]         ; Der Inhalt der Speicherstelle, deren Adresse in SI steht,
                    ; wird nach AX geladen. Die eckigen Klammern zeigen die
                    ; indirekte Adressierung an.
mov AX,[SI+4]       ; Der Inhalt der Speicherstelle, deren Adresse aus dem
                    ; Inhalt von SI + 4 berechnet wird, wird nach AX geladen.
mov AX,[SI+BX]      ; Der Inhalt der Speicherstelle, deren Adresse aus dem
                    ; Inhalt von SI + dem Inhalt von BX berechnet wird,
                    ; wird nach AX geladen.
mov AX,[SI+BX+4]    ; Der Inhalt der Speicherstelle, deren Adresse aus
                    ; dem Inhalt von SI + dem Inhalt von BX + 4 berechnet
                    ; wird, wird nach AX geladen.
```

Wie man sieht, können einige Zeigerwerte auch miteinander und mit Konstanten kombiniert werden. Das geht jedoch nicht beliebig mit allen Registern. Nur **BX** kann mit den anderen Zeigern **SI**, **DI** und **BP** kombiniert werden, nicht aber beispielsweise **SI** mit **DI**. **BX** kann aber auch allein oder mit einer Konstanten kombiniert werden. Diese Kombinationen sind sinnvoll zur Adressierung von Feldern. Natürlich ist nicht nur der Befehl `MOV` mit Zeigern zu kombinieren, sondern auch (fast) alle anderen Befehle, die mit Variablen arbeiten wie etwa `CMP`, `ADD`, `SUB`, `XCHG`, `AND`, `OR` und viele andere.

Im Protected Mode sind die Adressen länger, daher kommen dort die Zeiger **EDI**, **ESI**, usw. zum Einsatz.

Nachfolgend ist als Beispiel die Funktion `UPCASE` dargestellt, die alle Buchstaben eines Wortes in Großbuchstaben umwandelt. Nebenstehendes Flussdiagramm gehört dazu. Die Funktion prüft jedes Zeichen, ob es ein Kleinbuchstabe ist.[15] Wenn ja, wandelt sie ihn in einen Großbuchstaben um und setzt ihn an die gleiche Stelle in den String ein. Einen Rückgabewert im eigentlichen Sinne gibt es nicht,



[15] Aus Vereinfachungsgründen werden in diesem Beispiel keine Umlaute umgewandelt.

denn die Funktion kann ja dank des Zeigers auf den Stringanfang den String direkt bearbeiten.

Übergeben werden dem Unterprogramm

- in **CX**: die Anzahl der Buchstaben des Wortes.
- in **SI**: ein Zeiger auf das erste Zeichen des Wortes.

Hier das Listing dieser Funktion:

```
UPCASE:   ; Umwandlung eines Wortes in Großbuchstaben
          ; Übergabe Zeichenzahl in CX (für Zählschleife)
          ; Übergabe Zeiger auf Stringanfang in SI
  mov  AL,[SI] ; Zeichen aus String holen (SI zeigt darauf)
  cmp  AL,'a'  ; Zeichen in ASCII-Tabelle vor 'a'?
  jb   >L1     ; wenn ja, überspringen, keine Wandlung
  cmp  AL,'z'  ; Zeichen in ASCII-Tabelle nach 'z'?
  ja   >L1     ; wenn ja, überspringen, keine Wandlung
  and  AL,11011111xB ; Zeichen in Großbuchstaben wandeln
  mov  [SI],AL ; gewandeltes Zeichen in String einfügen
L1:
  inc  SI      ; Zeiger weiterstellen auf nächstes Zeichen
  loop UPCASE  ; Zählschleife, bis CX=0 ist
ret
```

17.2 Segment-Override

Für alle Befehle ist festgelegt, auf welches Segment sie sich beziehen. In den meisten Fällen ist dies das **Datensegment**, also die Segmentadresse, die in **DS** steht. Will man, dass sich ein Befehl **ausdrücklich auf ein anderes Segment** bezieht, schreibt man einfach das jeweilige Segmentregister mit einem anschließenden Doppelpunkt davor. Man spricht dann von *Segment-Override*. Es ist beim A386 jedoch auch möglich, den Segmentoverride direkt vor die betroffene Adresse zu schreiben.[16] Beispiele:

`ES:mov  B[020],10`	oder	`mov  ES:B[020],10`
`ES:and  AX,W[BX]`	oder	`and AX,ES:W[BX]`
`ES:cmp B[SI+BX+2],DL`	oder	`cmp ES:B[SI+BX+2],DL`
`CS:add W[MARKE+15],CX`	oder	`add ES:[MARKE+15],CX`

Anmerkung: Der Buchstabe `W` oder `B` beim 2. bis 4. Beispiel kann entfallen, da die Angabe eines Ziel- oder Quellregisters dem Compiler schon klar macht, ob ein Word oder ein Byte gemeint ist. Er erkennt das an der Registergröße.

Wir wollen uns diese Thematik einmal am Beispiel des Bildschirmspeichers ansehen. Der Bildschirmspeicher (im Textmodus) liegt an der Segmentadresse B800h. Für jedes

[16] Die meisten anderen Assembler erlauben dies nicht.

Zeichen auf dem Schirm wird ein Wort belegt, wobei das Zeichen im niederwertigen Byte liegt und die Farbe im höherwertigen Byte. Die vier niederwertigen Bit beschreiben dabei die Zeichenfarbe, die vier höherwertigen Bit die Hintergrundfarbe. Hierbei bestimmt das höchstwertige Bit, ob das Zeichen blinkt (Bit gesetzt) oder nicht (Bit gelöscht). Auch bei der Vordergrundfarbe hat das höchstwertige Bit (Bit 3) eine besondere Rolle; ist es gesetzt, ist die Intensität des Zeichens größer.

Sie finden nachfolgend das Listing des Programms `FARBTEST.ASM`.[17] Schauen Sie sich in diesem Beispiel im Bereich des Hauptprogramms an, wie aus dem Bildschirmspeicher gelesen und wie in ihn geschrieben wird. Was das Programm tut, steht im Kopf des Quellcodes. Besonders interessant ist die Routine `AUSGABE` ab Zeile 40 sowie das Lesen der aktuellen Farbe in Zeile 65 (markiert mit *2). Beachten Sie auch das Setzen des Segment-Registers **ES** in Zeile 60 bis 62 (markiert mit *1) ziemlich zu Anfang des Hauptprogramms.

```
; Programm FARBTEST.ASM zum Austesten der Bildschirmfarbe
; Die Nummer der Bildschirmfarbe wird in der untersten Zeile dargestellt.
; Mit Cursor-hoch und Cursor-runter wird die Nummer um 16 erhöht bzw. erniedrigt.
; Mit Cursor-rechts und Cursor-links wird die Nummer 1 erhöht bzw. erniedrigt.
; Die oberen 15 Zeilen werden in ihrer Farbe verändert.
jmp  START

;-- Variablendeklaration --------------------------
  FARBE1 db 0   ;  (Startwert, wird jedes mal geändert)
  FARBE2 db 04E ; Für die Ausgabe der Farbnummer, gelb auf rot
  TEXT:  db 'Änderung der Farbe mit den Cursortasten, Beenden mit <Esc>.',0D,0A,'$'
;-- Ende der Variablendeklaration -----------------
;-- Es folgen einige Unterprogramme ---------------
TASTATURABFRAGE:  ; Holt ein Zeichen von der Tastatur
  mov  AH,010
  int  016
ret
;-------------------------------------------------
HEXCODE:      ; Zeichen in AL in HEX-Code wandeln
              ; Das Zeichen wird der Routine in AL übergeben, die beiden
              ; Ziffern des Ergebnisses stehen dann in AH (das Zeichen für
              ; den höherwertigen Teil) und AL (das Zeichen für den
              ; niederwertigen Teil)
  mov  AH,AL  ; Wert auch nach AH kopieren
;-- den niederwertigen Teil in AL in ASCII-Zeichen wandeln:
  and  AL,0F  ; höherwertigen Teil in AL löschen
  add  AL,'0' ; niederwertigen Teil in ASCII-Zeichen wandeln
```

[17] Meine Schüler im BKT-Assembler-Kurs finden diese Datei auch im Austauschverzeichniss des Schulservers

```
  cmp  AL,'9' ; ist Zahl bis einschließlich Ziffer 9?
  jbe  >L1    ; wenn ja, fertig, überspringen
  add  AL,7   ; sonst noch Korrekturwert addieren
L1:
;-- den höherwertigen Teil in AH in ASCII-Zeichen wandeln:
  shr  AH,4   ; höherwertigen Teil in AH isolieren, nach unten schieben
  add  AH,'0' ; höherwertigen Teil in ASCII-Zeichen wandeln
  cmp  AH,'9' ; ist Zahl bis einschließlich Ziffer 9?
  jbe  ret    ; wenn ja, fertig
  add  AH,7   ; sonst noch Korrekturwert addieren
ret
;-----------------------------------------------------
AUSGABE:      ; Ausgabe der Farbnummer in letzter Bildschirmzeile:
  mov    AL,FARBE1      ; die aktuelle Farbnummer nach AL laden
  call   HEXCODE        ; Farbnummer in Hex-Code umwandeln
  push   AX             ; Ergebnis sichern
    mov    AL,AH        ; höherwertige ASCII-Zahl nach AL laden, soll zuerst
                        ;   ausgegeben werden
    mov    AH,FARBE2    ; die Farbe für die Ausgabe nach AH
    ES:mov W[160*24+2],AX ; in Bildschirmspeicher schreiben
                        ; (160*24 bedeutet: 80 Zeichen je 2 Byte pro Zeile
                        ;  mal 24 Zeilen. Nach 24 Zeilen, also in Zeile 25
                        ;  als 2. Zeichen soll das Zeichen stehen.)
  pop    AX             ; gesichertes Ergebnis (Hex-Code zurückholen)
  mov    AH,FARBE2      ; die Farbe für die Ausgabe nach AH
  ES:mov W[160*24+4],AX ; Ausgabe auf dem Bildschirm, 2 Bytes (entspr. ein
                        ;  Zeichen) weiter, hinter dem eben geschriebenen
ret
;-- Beginn des Hauptprogramms: ----------------------------------
START:
;-- Zuerst Textausgabe mit Anleitungstext:
  mov  DX,TEXT  ; Adresse des Textes
  mov  AH,09    ; Funktionsnummer Textausgabe
  int  021      ; Ausgabe durchführen
;-- Jetzt wird ES auf die Bildschirm-Segmentadresse eingestellt.
;   Da ES nicht direkt beschrieben werden kann, erfolgt der Umweg über AX.
  mov  AX,0B800     ; (*1)
  mov  ES,AX
;-- Die aktuelle Farbe des ersten Bildschirmzeichens oben links wird geholt
;   und in FARBE1 abgespeichert:
  ES:mov AL,[1]     ; Farbe des ersten Zeichens holen (in Byte Nr. 0 steht das
                    ;   Zeichen, dahinter die Farbe) (*2)
  mov    FARBE1,AL  ; Farbwert in Variable merken
;--
```

```
S1:  ; Hauptschleife, solange nicht <Esc> gedrückt wird
  call   AUSGABE    ; Ausgabe der aktuellen Farbnummer auf dem Bildschirm
  mov    CX,80*15   ; 80 Zeichen je Zeile, 15 Zeilen
  mov    DI,1       ; Schreib-Zeiger Di auf 1. Zeichen, 2. Byte
                    ; (Das HÖHERWERTIGE Byte stellt die Farbe dar. Es steht
                    ;  HINTER dem niederwertigen Byte.)
  mov    AL,FARBE1  ; AL auf Farbwert einstellen
  cld               ; Schreib-Richtung "vorwärts"
S2: ;-- Schleife. Ab hier werden 15 Zeilen gefärbt.
  stosb             ; Bildschirmzeichen färben und DI 1 weiterstellen
  inc    DI         ; Zeiger um noch ein Byte weiterstellen (insgesamt 2)
  loop   S2         ; CX runterzählen, solange noch nicht 0, weiter in Schleife
;-- Der obere Bildschirmteil ist jetzt eingefärbt.
  call   TASTATURABFRAGE
  cmp    AL,0E0
  je     SONDER
  cmp    AL,0
  jne    ANDERE
SONDER:
  cmp    AH,048     ; Taste Cursor-hoch?
  jne    >L1        ; wenn nein, überspringen
  add    FARBE1,010 ; Farbwert um 16 erhöhen
  jmp    S1         ; und zurück zum Schleifenanfang (Bildschirm färben)
L1:
  cmp    AH,050     ; Taste Cursor-runter?
  jne    >L1        ; wenn nein, überspringen
  sub    FARBE1,010 ; Farbwert um 16 erniedrigen
  jmp    S1         ; und zurück zum Schleifenanfang (Bildschirm färben)
L1:
  cmp    AH,04B     ; Taste Cursor-links?
  jne    >L1        ; wenn nein, überspringen
  dec    FARBE1     ; Farbwert um 1 erniedrigen
  jmp    S1         ; und zurück zum Schleifenanfang (Bildschirm färben)
L1:
  cmp    AH,04D     ; Taste Cursor-rechts?
  jne    >L1        ; wenn nein, überspringen
  inc    FARBE1     ; Farbwert um 1 erhöhen
  jmp    S1         ; und zurück zum Schleifenanfang (Bildschirm färben)
ANDERE:
L1:
  cmp    AL,01B     ; Taste <Esc> gedrückt?
  jne    S1         ; wenn nein, zum Schleifenanfang, ansonsten Ende
;-- Programmende:
  mov  AX,04C00
  int  021
```

18 Übungsaufgaben, Teil 4:

18.1 Aufgabe 4.1

Erstellen Sie das Programm `BLINKEIN.ASM`, das alle Zeichen, die auf dem Bildschirm zu sehen sind, blinken läßt. Dazu müssen Sie nur im Farbbyte jedes Zeichens das Blinkbit einschalten (höchstwertiges Bit im Farb-Byte, wie vorstehend beschrieben). Das Programm beendet sich sofort selbst. Erstellen Sie analog dazu das Programm `BLINKAUS.ASM`, mit dem das Blinken wieder abgeschaltet wird. **Anmerkung:** Je nach Einstellung der Grafikkarte funktioniert das Blinken nicht. Es erscheint dann nur eine andere Hintergrundfarbe. Möglicherweise funktioniert es auch nur im **Vollbildmodus**, unter Windows erreichbar mit <**Alt-Enter**>.

18.2 Aufgabe 4.2

Erstellen Sie das Programm `HALLO.ASM`. Das Programm löscht zunächst den Bildschirm mit einer ansprechenden Farbe. (Diese können Sie mit dem Programm FARBTEST aussuchen.) Danach schreibt es den Text „Hallo Welt“ an den Anfang der 10. Bildschirmzeile. Nach einem beliebigen Tastendruck beendet sich das Programm. Folgende Bedingungen sind zu beacheten:

- Der Text ist im Programm mit dem Zeichen 00h als Stringende-Zeichen abgeschlossen.
- Sämtliche Bildschirm-Operationen sollen ohne DOS- oder BIOS-Interrupts durch direkte Zugriffe auf den Bildschirmspeicher erfolgen.
- Das Löschen des Bildschirmes wird in einem Unterprogramm `BILDSCHIRM_LOESCHEN` ausgeführt. Dem Unterprogramm wird das **Löschzeichen in AL** und die **Farbe in AH** übergeben. Alle anderen Register sollen erhalten bleiben.
- Auch die Stringausgabe soll mit Hilfe eines Unterprogramms erfolgen. Es soll `AUSGABE` heißen. Dabei gelten folgende Übergabewerte an das Unterprogramm:
  - **SI**=Adresse des auszugebenden Strings
  - **CL**=Bildschirmspalte
  - **CH**=Bildschirmzeile
  - **AH**=Farbe des String
- Die Tastaturabfrage soll ebenfalls als Unterprogramm gestaltet sein. Sie können auf eine vorhandene Routine zurückgreifen.

Hinweis: Sie werden in der `AUSGABE`-Rouine vermutlich addieren und multiplizieren müssen, um die Offset-Adresse im Bildschirmspeicher zu berechnen. Machen Sie sich daher mit den Befehlen `MUL`, `IMUL`, `ADD`, `ADC`, `SUB` und `SBB` vertraut.

18.3 Aufgabe 4.3

Schreiben Sie ein Programm, das den Tastaturcode einer beliebigen Taste als **Dezimalcode** ausgibt. Nennen Sie das Programm `DTASTE`. Dazu können Sie das Programm, das den Code als Hexcode ausgibt, als Grundlage verwenden.[18] Sie müssen nur noch ein entsprechendes Unterprogramm (mit dem Namen `DEZCODE`) schreiben, das die Umrechnung vornimmt. Da das Ergebnis 3 Stellen haben kann, ist es sinnvoll, dass das Ergebnis in einem String abgelegt wird. Schließen Sie den String mit 00h ab, damit er mit dem Unterprogramm `AUSGABE` aus dem Programm `HALLO` (Aufgabe 4.2) ausgegeben werden kann. Sie müssen für den String also 4 Byte reservieren.

Folgende Bedingungen sollen für das Unterprogramm gelten:

- Die String-Adresse wird als Zeiger in **DI** übergeben.
- Alle anderen Register sollen erhalten bleiben.
- Das umzurechnende Zeichen wird in **AL** übergeben.

Sie können zur Ausgabe eine beliebige Farbe verwenden. Sie können auch selbst bestimmen, in welche Zeile der String geschrieben werden soll.

Ein Tip zur Umwandlung: Der in **AL** übergebene Wert wird durch 10 geteilt, der „Rest" ist die letzte Ziffer. Teilt man nun das Divisionsergebnis wieder durch 10, erhält man mit dem neuen „Rest" die vorletzte Ziffer, usw. Verwenden Sie einen **Zeiger**, um nach und nach die Werte in den String zu übertragen. Vergessen Sie dabei nicht, die Zahlen vorher in ASCII-Ziffern umzuwandeln. Da Sie hier dividieren müssen, machen Sie sich mit den Befehlen `DIV` und `IDIV` vertraut. Diese sind nicht ganz trivial in der Anwendung.

Wenn das Programm einwandfrei läuft, ändern Sie es dahingehend, dass nun nicht nur **AL** sondern das ganze **AX**-Register in Dezimalcode umgewandelt wird. Das Programm heißt nun `DTASTE2`. (Zugegebenermaßen hat dieses Programm keine wirkliche praktische Bedeutung, es dient ausschließlich Übungszwecken.)

[18] Sie haben dieses Programm mit dem Namen HTASTEN als Aufgabe 3.3 geschrieben.

18.4 Aufgabe 4.4

Das **BIOS** eines jeden Rechners benötigt für die Verwaltung seiner Aufgaben Platz im RAM für seine Daten. Diese Daten liegen immer im **BIOS-Datensegment**. Das BIOS-Datensegment beginnt an der **Segment-Adresse 40hex**. Drückt beispielsweise der Anwender auf irgendeine Taste, dann löst der Interrupt-Controller den **Hardware-Interrupt 09** aus. Das momentan aktive Vordergrundprogramm wird unterbrochen. Diese Interrupt-Routine fragt die Tastatur ab, welche Taste es denn war. Sie schaut dann in einer Tabelle nach, ob aktuell die Shift-Taste, die Strg-Taste oder eine ähnliche Taste gedrückt ist. In Abhängigkeit davon schaut sie in einer anderen Tabelle nach, welches Zeichen damit gemeint ist. Dieses Zeichen muss nun im Tastaturpuffer zur Abholung durch den Software-Interrupt 16hex bereitgestellt werden. Dieser Tastaturpuffer befindet sich im BIOS-Datensegment, wo er auch anderen BIOS-Funktionen verfügbar ist.

Da es sein kann, dass mehrere Tasten betätigt werden, bevor das Vordergrundprogramm Zeit findet, die Tastatur über den Interrupt 16hex abzufragen, bietet der Tastaturpuffer für bis zu 16 Zeichen Platz. Verwaltet wird er mit zwei Zeigern, einem Schreib- und einem Lesespeicher. An die Schreibzeiger-Position wird das eingegebene Zeichen geschrieben. Nach dem Schreiben des Zeichens wird der Schreibzeiger um 2 Stellen (jedes Zeichen belegt ein *word*) erhöht. Hat der Schreibzeiger das obere Ende des Pufferbereiches erreicht, dann wird er nicht weiter erhöht, sondern auf den Anfang des Pufferbereiches eingestellt.

Sinngemäß das gleiche passiert mit dem Lesezeiger im Interrupt 16hex, wenn ein Zeichen abgeholt wird. Zeigen Schreib- und Lesezeiger auf die gleiche Stelle, dann gilt der Puffer als leer, obwohl natürlich die letzten Zeichen noch im RAM stehen. Das Spiel der Zeiger im Tastaturpuffer wollen wir mit unserem nächsten Übungsprogramm beobachten.

Ab Offset **1Ehex** bis **3Dhex** liegt der Bereich des **Tastaturpuffers**. Jedes Zeichen, das in diesem Puffer abgelegt ist, besteht aus 2 Byte (also Wortbreite), wobei die Zeichen so aufgebaut sind, wie bei der Ausgabe über den Tastatur-Interrupt **16hex**. Der zugehörige Lesezeiger liegt bei der Offset-Adresse **1Ahex**, der Schreibzeiger bei **1Chex**, also unmittelbar **vor** dem eigentlichen Pufferbereich.

Schreiben Sie ein Program mit dem Namen `TPUFFER` mit folgenden Eigenschaften: Das Programm soll den Zustand und den Inhalt des Tastaturpuffers anzeigen. Dazu wird zunächst der Bildschirm gelöscht. Anschließend wird eine kleine Anleitung ausgegeben und eine Art Überschrift für die nachfolgenden Zahlenkolonnen geschrieben. Das Bild sieht dann zunächst so aus:

```
Bitte eine Taste drücken!
Bei jedem Tastendruck wird der aktuelle Inhalt des Tastaturpuffers dargestellt.
Mit der Taste <Esc> wird das Program beendet.
 Zeiger                            Daten
!------! !--------------------------------------------------------------!
```

Mit jedem Tastendruck wird nun der **Inhalt des Tastaturpuffers** in einer eigenen Zeile dargestellt. Die Zeichen liegen im Puffer als Worte, wie oben beschrieben. Stellen Sie daher jedes Wort als **Hex-Code** dar. Zwischen den Worten bleibt kein Platz, sonst wird die Ausgabezeile zu lang. **Vor** dem Pufferinhalt werden die Inhalte des Lese- und des Schreib-Zeigers dargestellt. Mit der Taste <**Esc**> wird das Programm beendet.

Nach der Eingabe einiger Zeichen sieht das Bild etwa folgendermaßen aus:

```
Bitte eine Taste drücken!
Bei jedem Tastendruck wird der aktuelle Inhalt des Tastaturpuffers dargestellt.
Mit der Taste <Esc> wird das Program beendet.
 Zeiger                            Daten
!------! !--------------------------------------------------------------!
003A003A 39203920392039201C0D1C0A2B3E39200B300B30342E0B301C0D1E4139203920
003C003C 39203920392039201C0D1C0A2B3E39200B300B30342E0B301C0D1E4130423920
001E001E 39203920392039201C0D1C0A2B3E39200B300B30342E0B301C0D1E4130422E43
00200020 20443920392039201C0D1C0A2B3E39200B300B30342E0B301C0D1E4130422E43
00220022 20441245392039201C0D1C0A2B3E39200B300B30342E0B301C0D1E4130422E43
00240024 204412451C0D39201C0D1C0A2B3E39200B300B30342E0B301C0D1E4130422E43
```

Wie sind diese Angaben zu interpretieren? Unter der Überschrift `Zeiger` stehen Schreib- und Lesezeiger des Tastaturpuffers. In der ersten Zeile haben beide den Wert 003Ahex. Gleiche Werte beider Zeiger bedeuten, dass zum Ablesezeitpunkt der Tastaturpuffer leer war. Das nächste Zeichen wird dann an die Offset-Adresse 003Ahex geschrieben.

In der nächsten Zeile ist genau das passiert. Neben den Zeigern ist unter der Überschrift `Daten` der Inhalt des Tastaturpuffers dargestellt. Die acht- bis fünft-letzte Ziffer hat sich in der zweiten Zeile auf 3042hex geändert. Das ist genau der Zeichencode, der mit dem Tastatur-Interrupt ausgelesen wird, wenn ein „**B**" eingegeben wurde. Hier wurde er vom Tastaturtreiber (Interrupt 09) abgelegt und hier wurde er auch vom Tastatur-Interrupt (Interrupt 16hex) ausgelesen. Die beiden Zeiger zeigen nun auf das **letzte *Word*** im Pufferbereich, wo das nächste Zeichen abgespeichert wird (siehe nächste Zeile). Ist das geschehen, wird der Pufferbereich wieder von vorn beschrieben, usw.

Es ist sicher sinnvoll, wenn Sie einige bereits erstellte Unterprogramme übernehmen, beispielsweise zur Umwandlung eines *Byte*-Wertes in einen String, der den Hex-Code angibt, oder zur Tastaturabfrage.

Machen Sie sich mit den Befehlen für String-Operationen `LODSB`, `LODSW`, `STOSB` und `STOSW` vertraut.

19 Diverses

19.1 Hardwarenahe Befehle

Ein Rechner beinhaltet allerlei spezielle Baugruppen wie Interrupt-Controller, Tastatur-Controller oder Timer-Baustein. Diese Baugruppen sind durch einen **Bus** mit dem Prozessor verbunden. Auch zusätzlich eingebaute Steckkarten wie die Grafikkarte odes ähnliches hängen an diesem Bus. Will man eine dieser Baugruppen ansprechen, muss im Adress-Teil dieses Busses die Adresse der jeweiligen Baugruppe eingestellt werden. Dann können über den Daten-Teil des Busses Daten zwischen Prozessor und Baugruppe ausgetauscht werden. Machen Sie sich mit den Befehlen `IN` und `OUT` vertraut. Diese beziehen sich auf die Komunikation über mit diesen Baugruppen.

Als Beispiel wollen wir uns einmal die Tonerzeugung mit dem **Timer-Baustein 8253** und dem **Peripheriebaustein 8255** ansehen. Um dem 8253 einen bestimmten Ton zu entlocken, muss in das Register mit der Portadresse **42hex** ein 16-Bit Wert geschrieben werden, durch den die **interne Taktfrequenz** geteilt wird. Diese Frequenz liegt bei **1.193.180 Hz**. Das Divisionsergebnis gibt die Frequenz des zu erzeugenden Rechtecksignals an, das die Lautsprechermembran zum Schwingen bringt. Allerdings kann dieser Wert nicht „am Stück“ als ganzes *word* übertragen werden, dies geschieht **byteweise**. Zuerst muss das niederwertige Byte übertragen werden und anschließend das höherwertige (an die gleiche Portadresse!). Auf diese Weise steht danach der Wert richtig im Register des 8253. Vor der Übergabe des Wertes für die Frequenzteilung muss jedoch das **Steuerregister des 8253** über die Portadresse **43hex** mit dem Wert **182** (=**B6hex)** geladen werden, um den Baustein für den Lautsprecherbetrieb zu initialisieren.

Der **Peripheriebaustein 8255** steuert das ganze. Er verfügt über drei parallele 8-Bit-Schnittstellen. Zwei sind für die Eingabe und eine für die Ausgabe von Daten zuständig. Die Register für die Eingabeschnittstellen besitzen die Port-Adressen **60hex** und **62hex**, das Register für die Ausgabeschnittstelle wird über die Portadresse **61hex** angesprochen. Die **untersten beiden Bits** des letzten Registers steuern den PC-Lautsprecher. Ist das **Bit 0** dieses Registers auf **1** gesetzt, dann wird der **zweite Kanal des Timerbausteins 8253** dazu veranlasst, ein Rechtecksignal zu erzeugen, dessen Frequenz wie oben beschrieben eingestellt wurde. Ist auch **Bit 1** des Registers an der **Portadresse 61hex** auf **1** gesetzt, so wird das Rechtecksignal des 8253 **an den Lautsprecher angeschlossen**, ein Ton wird erzeugt. Zum Abschalten des Tones muss nur **Bit 1** wieder auf **0** gesetzt werden, der Lautsprecher ist dann wieder von der Tonerzeugung getrennt. **Alle anderen Bits sollte man auf dem ursprünglichen Wert belassen!**

19.2 Interrupts umleiten

Eine weitere wichtige Anwendung für die IO-Befehle ist die Kommunikation mit der Tastatur. Wird auf der Tastatur eine Taste betätigt, löst der Interruptcontroller einen **Interrupt 09hex** aus, wie bereits bei Aufgabe 4.4 beschrieben. Dieser wird vom Tasta-

turtreiber benutzt, um festzustellen, welche Taste gedrückt wurde und welches Zeichen demnach im Tastaturpuffer abgelegt werden soll. Dieses kann dann (bekanntlich) über den **Interrupt 016hex** abgefragt werden.

Wenn man Interrupts auf die eigene Routine umleiten will, muss man eine Reihe von Dingen beachten, damit es nicht zum Absturz kommt:

- Bei Beendigung des Programms muss der Interrupt unbedingt wieder auf den alten Wert zurückgestellt werden, es sei denn, es handelt sich um ein TSR-Programm. (Ein TSR-Programm bleibt auch nach „Beendigung" zumindest teilweise aktiv im Speicher zurück.) Der Tastaturtreiber ist beispielsweise ein TSR-Programm.
- Man muss sich im klaren sein, ob die eigene Routine eine fremde **ersetzen** oder **ergänzen** soll.
  - Ist es ein **Ersatz**, dann endet die Routine mit einem `IRET` (nicht mit `RET` oder `RETF`!), wodurch nicht nur zum nächsten Befehl im unterbrochenen Programm zurückgesprungen wird, sondern auch die Flags vom Stapel genommen werden.
  - Beim **Ergänzen** kann die eigene Routine vor oder hinter der Original-Routine abgearbeitet werden.
    * Wird sie **vorher** durchlaufen, schließt sie mit einem `FAR JUMP` an die ursprüngliche Interruptadresse ab. Das `IRET` am Ende der ursprünglichen Routine sorgt dann für den ordnungsgemäßen Rücksprung an die Stelle, wo der Interrupt auftrat.
    * Ein wenig trickreicher muss man arbeiten, wenn zuerst die ursprüngliche Routine ablaufen lassen will und erst danach die eigene. Ein einfacher `FAR CALL` an die ursprüngliche Adresse reicht nicht aus, denn das `IRET` am Ende der Ursprungsroutine nimmt nicht nur die Rücksprungadresse vom Stapel, sondern auch noch ein weiteres Wort, was in das Flag-Register geladen wird. Die Original-Routine geht ja von einem Aufruf durch `INT` aus, wodurch zuerst das Flag-Register auf den Stapel gebracht wird. Die Vorgehensweise sieht dann so aus: Der erste Befehl der eigenen Routine lautet `PUSHF`. Damit wird der Inhalt des Flagregisters auf den Stapel gebracht. Es folgt ein `FAR CALL` an die Adresse des alten Interrupts. Dieser wird ausgeführt, und das `IRET` an seinem Ende nimmt die Flags vom Stapel, die wir zuvor dort abgelegt haben. Der Rücksprung erfolgt dann an unsere eigene Routine, an die Stelle hinter dem `FAR CALL`. Unsere eigene Routine schließt dann mit einem `IRET` ab.
- Da insbesondere bei Hardware-Interrupts nie klar ist, von wo aus (Programmstelle) sie aufgerufen worden sind, müssen alle Register, die man verwenden will, auf dem Stapel gesichert und am Schluss zurückgespeichert werden, damit das unterbrochene Programm ordnungsgemäß weiterarbeiten kann. Dabei muss man sich darüber

im klaren sein, dass man dabei möglicherweise einen „fremden“ Stapel benutzt, dessen Größe man nicht kennt. Will man viele Werte auf dem Stapel ablegen, ist es besser, vorher den Stapel auf einen Bereich im eigenen Programm umzustellen. Nachher das Zurückstellen nicht vergessen!

- Auch die Segment-Register (mit Ausnahme von **CS** natürlich) können beim Eintritt in unsere Routine jeden beliebigen Wert haben. Will man auf Daten im RAM zugreifen, dann muss das entsprechende Segmentregister (meist **DS**) erst auf den richtigen Wert eingestellt werden. Bitte nicht vergessen, **DS** vorher zu sichern und am Schluss zurückzuspeichern! Wie man sieht, sind viele Dinge zu beachten, die in „normalen“ Programmen nicht auftreten. Dabei liegt der Teufel oft im Detail. Ein Beispiel dazu:

Sie wollen Ihre Routine vor der ursprünglichen ablaufen lassen, müssen also als Abschluss ein `FAR JUMP` ausführen. Die Adresse ist zum Zeitpunkt der Erstellung des Programmes noch nicht bekannt, sie wird ja erst in der Laufzeit ermittelt. Nun kann man natürlich mit dem Befehl `ALTER_INTERRUPT DD 0` ein Doppelwort reservieren und die Variable zur Laufzeit mit dem richtigen Wert belegen. Ein `JMP ALTER_INTERRUPT` würde leider nur dann funktionieren, wenn **DS** auf das richtige Datensegment zeigen würde. Leider müssen wir **DS** zuvor auf den alten Wert zurückstellen! Es nützt uns also nichts, dass der Compiler an dem `DD` der Variablendeklaration erkennt, dass ein `FAR JUMP` ausgeführt werden soll. Er würde die Adrese in dem Datensegment suchen, das zu **DS** passt. Wird die nicht berücksichtigt, kann es sein, dass unser Programm 20 mal reibungslos läuft und beim 21. Mal abstürzt, weil dann vielleicht gerade der Interrupt bei der Bearbeitung eines anderen Interruptes (mit anderem Datensegment) auftrat. Es gibt 2 Abhilfemöglichkeiten.

1. Wir setzen ein `CS:` vor den Sprungbefehl. Wenn unser Datensegment mit unserem Codesegment übereinstimmt, wird der Segment-Override klappen.
2. Etwas trickreicher ist die andere Variante, die nicht nur Speicherplatz spart, sondern auch schneller abläuft. Man kann die Adresse nämlich einfach „einpatchen“. Das geht so: Man schreibt im Programm zunächst einen `JMP`-Befehl mit einem Platzhalter, etwa `JMP 0:0`. An der doppelten 0 mit dem Doppelpunkt dazwischen erkennt der Compiler den `FAR JUMP`. Die Adresse ist natürlich zunächst falsch, sie muss zur Laufzeit des Programms geändert werden. Im Maschinencode belegt der Befehl 5 Byte, wobei im ersten Byte **EAhex** steht (das ist der Sprungbefehl) und dahinter als Doppelwort die Sprungadresse (zunächst 0:0). Man muß also nur noch die beiden Worte ein, bzw. 3 Byte hinter dem Anfang des Sprungbefehles einschreiben, als ob dort Daten stünden. Damit ändert sich der Befehl entsprechend ab. Das sieht dann etwa so aus:

```
ADRESSE:
 jmp  0:0
; später dann etwa:
 CS:mov W[ADRESSE+1],AX
 CS:mov W[ADRESSE+3],BX

;-- Alternative:
 jmp  0:0
ALTER_INT equ $-4
; später dann etwa:
 CS:mov W[ALTER_INT ],AX
 CS:mov W[ALTER_INT+2],BX
```

Anhand des Labels `ADRESSE` kann die korrekte Speicherstelle berechnet werden, an der die beiden Worte eingeschrieben werden sollen. Im Beispiel wird der niederwertige Teil der Sprungadresse aus **AX** *1 Byte hinter den Anfang des Sprungbefehls* übertragen und der höherwertige Teil aus **BX** *3 Byte hinter seinen Anfang*.

Der gleiche Trick funktioniert übrigens sinngemäß auch mit dem `CALL`-Befehl, nur dass das erste Byte jetzt den Code **9Ahex** enthält. Das ist der Code für den `FAR CALL`.

Ein weiteres Problem: Will oder muss man den Stapel auf die eigene Routine umstellen, dann muss auf jeden Fall verhindert werden, dass während der Abarbeitung der Interruptroutine diese erneut aufgerufen werden kann – ein Absturz wäre kaum mehr vermeidbar. Der Grund liegt darin, dass beim erneuten Umstellen des Stapels auf einen eigenen Bereich genau der Bereich überschrieben würde, aus dem heraus der aktuelle Interrupt auftrat! Die Daten aus dem ersten Interrupt wären damit durch falsche Werte überschrieben und verloren. Da man grundsätzlich nie weiß, wann ein hardwaregesteuerter Interrupt ausgelöst wird, kann das natürlich auch passieren, während die eigene Routine aktiv ist.

Eine Abhilfe sieht etwa so aus: Bevor der Stapel umgestellt wird, wird ein Flag geprüft, das man im eigenen Datenbereich angelegt hat. Dieses Flag wird gesetzt unmittelbar bevor der Stapelbereich umgestellt wird. Natürlich muss es nach Zurückstellen des Stapels wieder gelöscht werden. Ergibt die Prüfung dieses Flags beim Eintritt in die Ersatzroutine, dass es gesetzt ist, darf die Ersatzroutine nicht weiter ausgeführt werden. Je nach Art des Interrupts wird er nun einfach mit einem `IRET` beendet, oder es erfolgt ein Sprung an die ursprüngliche Interruptadresse. Nachfolgend sehen Sie einen Ausschnitt aus dem entsprechenden Programmteil.

```
  CS: cmp   FLAG,0 ; Flag gesetzt?
  je  >L1          ; wenn nein, überspringen
  jmp ALTER_INT    ; (oder IRET)
L1:
  CS: mov FLAG,1   ; Flag setzen
                   ; (Hier steht die eigene Routine)
  CS: mov  FLAG,0  ; Flag zurücksetzen
  jmp ALTER_INT    ; (oder IRET)
```

Wie wird der Stapel umgestellt? Auch hierbei gibt es mögliche Fallen, in die man nicht stolpern sollte. Logisch ist es sicher, dass man sich zunächst einen Bereich im Codesegment schafft (oder geschickt aussucht), der hierfür groß genug ist. In den meisten Fällen reicht die **DTA** (Disk-Transfer-Area) zwischen Byte **80hex** und **100hex** im **PSP** (Programm-Segment-Prefix) aus. Bei **COM**-Dateien liegt der **PSP** am Anfang des Code-Segmentes. Dieser Bereich wird normalerweise nur für Übergabeparameter an das Programm verwendet. Geschieht dies nicht, oder sind die Übergabewerte schon ausgewertet, ist der Bereich frei. Da der Stapel „nach unten" wächst, ist der Startwert für **SP** die höchste Wortadresse am Ende des vorgesehenen Bereiches, in unserem Beispiel also **100hex**. Hier ist eine mögliche Sequenz dargestellt, mit der der Stapel umgstellt werden kann.

```
CS: mov  ALT_SS,SS  ; altes Stacksegment merken
CS: mov  ALT_SP,SP  ; alten Stapelzeiger merken
cli                 ; Interrupt sperren!
push  CS            ; Das Codesegment in das
pop SS              ;   Stacksegment übertragen
mov SP,STAPELANFANG ; Stapelzeiger auf Startwert
sti                 ; Interrupts wieder freigeben
                    ; Hier alle zu sichernden Register
                    ;  mit push auf den Stapel bringen.
push CS             ; Das Codesegment in das
pop DS              ;   Datensegment übertragen
```

Hierbei stellt sich eine wichtige Frage: Was passiert, wenn ausgerechnet in den Augenblick ein Interrupt auftritt, wenn das Stacksegment schon umgestellt ist, aber noch nicht der Stackpointer? Das wäre in Zeile 5 in nebenstehender Sequenz. Dann passen die Werte nicht zusammen, ein Absturz wäre die unvermeidliche Folge. Deshalb ist es **zwingend notwendig**, während der kritischen Phase alle Interrupts zu sperren. Die geschieht mit `CLI` vorher, und das nachfolgende `STI` erlaubt wieder die Interrupts. Vergisst man diese Vorsichtsmaßnahme, kann es durchaus sein, dass es 1000 mal gut geht und beim 1001. Mal nicht. Unser Programm wäre ähnlich instabil, wie ein mehr oder weniger bekanntes Betriebssystem...

Das Zurückstellen des Stapels geht ähnlich, nur in umgekehrter Reihenfolge. Nachfolgendes Beispiel zeigt eine Möglichkeit. **Auch hier ist wieder darauf zu achten, dass während der eigentlichen Umstellung die Interrupts gesperrt sind!**

```
                   ; zuerst alle alten Register-Werte mit pop
                   ;   zurückholen und dann:
cli                ; Interrupt sperren!
CS: mov SP,ALT_SP  ; SP wiederherstellen
CS: mov SS,ALT_SS  ; SS wiederherstellen
sti                ; Interrupts wieder freigeben
```

20 Übungsaufgaben, Teil 5

20.1 Aufgabe 5.1

Untenstehend finden Sie das Programm `TON.ASM` im Quellcode.[19] Das Programm soll einen Ton ausgeben, dessen Frequenz und Länge dem Programm als Parameterstring übergeben wird.

Beispiel: Die Eingabe `TON 800 500` soll bewirken, dass ein Ton mit einer Frequenz von **800Hz** und einer Länge von **500ms** erzeugt werden soll. Danach beendet sich das Programm.

Das Programm ist in Teilen schon fertig. Was noch fehlt, sind die beiden Unterprogramme `TONAUSGABE` und `TON_AUS`. In Kommentarzeilen steht, was diese Unterprogramme tun sollen. Füllen Sie die Lücken in den Unterprogrammen mit Code, die die entsprechenden Funktionen bewirken. Verwenden Sie dazu die Befehle `IN` und `OUT` und beachten Sie die vorstehenden Ausführungen über die Ansteuerung des Timers 8253 und des Preipheriebausteins 8255. Sie sollten sich im bereits fertigen Teil des Programms auch einmal genau ansehen, wie auf die Übergabe-Parameter der Kommandozeile zugegriffen wird.

Hier noch zwei Hinweise:

- Unter Windows funktioniert die Tonerzeugung nur im **Vollbildmodus**, den Sie mit <**Alt-Enter**> erreichen.
- Testen Sie bitte nicht die Tonerzeugung, bevor das Unterprogramm `TON_AUS` fertig ist! **Auch nach Beendigung des Programms bleibt nämlich ein eventuell eingeschalteter Ton aktiv!**

Hier folgt das Listing des (unvollständigen) Programms `TON`:

```
; Programm TON
; Das Programm erzeugt einen Ton, dessen Frequenz und Länge im
; Parameterstring übergeben wird. Beispiel: TON 800 500
; Im Beispiel würde ein Ton von 800Hz mit einer Länge von 500ms erzeugt.
  jmp  START
;-- Diverse Texte --------------------------------
ANLEITUNGSTEXT:
db 'Bitte dem Programm zwei Zahlenwerte für Frequenz (in Hertz) und',0A,0D
db 'Länge (in Millisekunden) übergeben. Beispiel für 800Hz und 500ms:',0A,0D
db 'TON 800 500',0A,0D,'$'
;--------------
FREQUENZFEHLER:
db 'Die als 1. Parameter angegebene Frequenz ist zu hoch!',0D,0A,'$'
;--------------
```

[19] Meine Schüler im BKT-Assembler-Kurs finden diese Datei auch im Austauschverzeichniss des Schulservers

```
ZEITZFEHLER:
db 'Die als 2. Parameter angegebene Zeit ist zu groß!',0D,0A,'$'
;-- Variablen: ------------------------------------
FREQUENZ dw 0
LAENGE   dw 0
;====================================================
TONAUSGABE: ; schaltet Ton ein, dessen Frequenz in Hz in AX übergeben wird
;-- Zuerst wird der Teiler aus der Frequenz berechnet.
;   Dazu muß die Taktfrequenz von 1193180 (= 1234DC hex) durch die
;   übergebene Frequenz geteilt werden. Das Ergebnis ist der Teiler,
;   der dem Timerbaustein 8253 übergeben werden muß.




;-- Es folgt die Tonausgabe. Dazu zuerst Ausgaberegister für Timerbaustein
;   8253 vorbereiten:




;-- das niederwertige Byte übermitteln:

;-- das höherwertige Byte übermitteln:


;-- alten Status des 8253 holen:

;-- und den Lautsprecher einschalten:


ret
;------------------
TON_AUS:  ; Den Ton wieder abschalten
;-- alten Status des 8253 holen:

;-- und den Lautsprecher ausschalten:


ret
;------------------
```

```
WARTEN:    ; Wartet Zeit ab, die in AX (in ms) übergeben wird
  mov  CX,183        ; Anzahl der Schleifendurchläufe
  mul  CX            ;  = Zeit (in ms) * 183/10000
  mov  CX,10000
  div  CX
  mov  SI,AX         ; Zahl der notwendigen Schleifendurchläufe nach SI
  inc  SI            ; nach oben aufrunden
  mov  AX,0
  int  01A           ; Zahl der Timerticks holen
  mov  DI,DX         ; Zählerstand in DI merken
  mov  CX,SI         ; Zähler CX für Zählschleife setzen
S1:                  ; Warteschleife  1 Timertick
  push CX
    mov  AX,0
    int  01A         ; Zahl der Timerticks holen
  pop  CX
  cmp  DX,DI         ; wurde inzwischen hochgezählt?
  je   S1            ; wenn nein, kleine Warteschleife
  mov  DI,DX         ; neuen Wert in DI merken
  loop S1            ; Zählschleife
ret
;-------------------
WARNTON:           ; Ausgabe eines Trillers als Warnton:
  mov  AX,670      ; Frequenz 670 Hz
  call TONAUSGABE
  mov  AX,100      ; Zeit 100 ms
  call WARTEN
  mov  AX,800      ; Frequenz 800 Hz
  call TONAUSGABE
  mov  AX,100      ; Zeit 100 ms
  call WARTEN
  mov  AX,670      ; Frequenz 670 Hz
  call TONAUSGABE
  mov  AX,100      ; Zeit 100 ms
  call WARTEN
  call TON_AUS     ; und Ton wieder abschalten
ret
;-------------------
```

```
PRUEFE_ZIFFER: ; prüft, ob Zeichen in AL eine Ziffer ist.
               ; wenn ja, wird das ZF gesetzt, anderenfalls gelöscht.
  cmp  AL,'0'
  jb   ret    ; wenn AL kleiner '0', dann fertig mit gelöschtem ZF.
  cmp  AL,'9'
  ja   ret    ; wenn AL größer '9', dann fertig mit gelöschtem ZF.
  test AL,0C0 ; Zero-Flag setzen (die höchsten Bits von AL sind glöscht!)
ret
;-----------------
PRUEFE_PARAMETER: ; prüft, ob 2 Parameter mit Ziffern übergeben wurden
                  ; wenn nein, wird das CF gesetzt, sonst gelöscht.
  mov  SI,080     ; Zeiger auf Parameterstring-Anfang einstellen
  cld             ; Richtungsflag für "vorwärts"
;-- Die Stringlänge wird nach CX geholt:
  lodsb           ; erstes Zeichen nach AL holen
  mov  CL,AL      ; Stringlänge nach CX laden
  mov  CH,0       ; (Dazu muß CH=0 sein)
  mov  DL,0       ; DL wird als Zähler für die Strings verwendet - Startwert
  jcxz >F0        ; wenn CX=0, dann fertig
;--
S1:
  lodsb              ; nächstes Zeichen holen
  call PRUEFE_ZIFFER ; prüft, ob Zeichen in AL eine Ziffer ist.
  loopne S1          ; CX runterzählen - Wenn keine Ziffer, weiter in Schleife
;-- Anfang von Ziffernstring gefunden --
  inc  DL            ; Zählen
  jcxz   >F0         ; wenn Stringende erreicht, zum Ende springen
S2:
  lodsb
  call PRUEFE_ZIFFER ; prüft, ob Zeichen in AL eine Ziffer ist.
  loope S2
  jcxz  >F0
  jmp   S1
;---------
F0:                  ; Ende
  cmp  DL,2          ; mindestens 2 Zahlen gefunden?
  jae  >F1           ; wenn ja, Sprung (fertig ohne Fehler)
  stc                ; "Fehler" markieren
ret
;--
F1:
  clc                ; "Kein Fehler" markieren
ret
;-------------------
```

```
STRING_IN_ZAHL: ; wandelt String bei DS:SI in Zahl
                ; Die Zahl wird in AX zurückgegeben
                ; Ist die Zahl zu groß, wird CF gesetzt.
  mov  CX,10    ; Faktor zum multiplizieren
  xor  AX,AX    ; Startwert: AX=0
  mov  BH,0
  cld           ; Richtungsflag für "vorwärts"
S1:
  xchg AX,BX    ; vorübergehender Tausch AX mit BX (wegen nachf. Befehl LODSB)
  lodsb         ; ein Zeichen holen
  call PRUEFE_ZIFFER ; prüft, ob Zeichen in AL eine Ziffer ist.
  xchg AX,BX    ; AX mit BX zurücktauschen
  jne  >L2      ; wenn keine Ziffer, dann fertig
  mul  CX       ; bisherigen Zahlenwert mal 10 nehmen
  cmp  DX,0     ; gibt es einen höherwertigen Teil, der nicht in AX passte?
  je   >L1      ; wenn nein, überspringen
  stc           ; sonst Fehler markieren
ret             ; und fertig
;--
L1:
  sub  BL,'0'   ; ASCII-Zeichen in Zahl wandeln
  add  AX,BX    ; aktuelle Ziffer zum Zahlenwert dazu addieren
  jc   ret      ; wenn jetzt ein Überlauf passierte, Abbruch mit Fehler
  jmp  S1
L2:
  clc           ; CF löschen, da Funktion ohne Fehler ablief
ret
;------------------------------------------------------------------
START:
  call PRUEFE_PARAMETER ; prüft, ob 2 Parameter mit Ziffern übergeben wurden
  jc   ANLEITUNG        ; Im Fehlerfall Anleitung ausgeben und fertig
;-- Anfang des ersten Zahlenstrings suchen:
  mov  SI,081        ; Zeiger auf Anfang des Parameter-String einstellen
  cld                ; Richtungsflag für "vorwärts"
S1:
  lodsb              ; nächstes Zeichen holen
  call PRUEFE_ZIFFER ; prüft, ob Zeichen in AL eine Ziffer ist.
  jne  S1            ; wenn nein, weitersuchen in Schleife
```

```
;-- Anfang des ersten Zahlenstring gefunden - Umwandeln!
  dec  SI             ; Zeiger zurück auf den Anfang des Zahlenstring
  call STRING_IN_ZAHL ; wandelt String bei DS:SI in Zahl (in AX)
  jc   FREQUENZ_ZU_GROSS ; bei Überlauf mit Fehlermeldung abbrechen
  mov  FREQUENZ,AX    ; Frequenzwert abspeichern
;-- Anfang des zweiten Zahlenstring suchen:
S1:
  lodsb               ; nächstes Zeichen holen
  call PRUEFE_ZIFFER ; prüft, ob Zeichen in AL eine Ziffer ist.
  jne  S1             ; wenn nein, weitersuchen in Schleife
;-- Anfang des zweiten Zahlenstring gefunden - Umwandeln!
  dec  SI             ; Zeiger zurück auf den Anfang des Zahlenstring
  call STRING_IN_ZAHL ; wandelt String bei DS:SI in Zahl (in AX)
  jc   ZEIT_ZU_GROSS
  mov  LAENGE,AX      ; Zeitdauer abspeichern
;-- Frequenz und Dauer sind bestimmt. Jetzt kann der Ton ausgegeben werden!
  mov  AX,FREQUENZ    ; Frequenz (in Hertz) für die Tonausgabe laden
  call TONAUSGABE
  mov  AX,LAENGE      ; Zeit (in Millisekunden) für Tondauer laden
  call WARTEN
  call TON_AUS        ; und Ton wieder abschalten
  jmp  ENDE           ; das wars schon!
;-- Es folgen die Textausgaben für die jeweiligen Fehler:
FREQUENZ_ZU_GROSS:
  mov  DX,FREQUENZFEHLER
  jmp  AUSGABE
;--
ZEIT_ZU_GROSS:
  mov  DX,ZEITZFEHLER
  jmp  AUSGABE
;--
ANLEITUNG:
  mov  DX,ANLEITUNGSTEXT
AUSGABE:
  push DX
    call WARNTON
  pop  DX
  mov  AH,09          ; Text ausgeben
  int  021
ENDE:
  mov  AH,04C
  int  021
```

20.2 Aufgabe 5.2

Erstellen Sie das Programm mit dem Namen `SCANCODE` mit folgenden Eigenschaften:

Das Programm stellt den **Scancode** jeder Taste (im Hexcode) dar und schreibt dahinter das dabei entstandene Zeichen sowie dessen Hexcode. Der Scancode ist der Code, den die **Tastatur** erzeugt, nicht der Zeichencode des zugehörigen Buchstabens. Ein Scancode wird sowohl beim Betätigen, als auch beim Loslassen einer Taste erzeugt. **Alle** Tasten erzeugen einen Scancode, auch die Shift-Taste, die Strg-Taste usw.

Als Hilfe zur Programmerstellung finden Sie einen bereits halb fertigen Entwurf im Anschluss an diese Programmbeschreibung.[20] Das Programm besteht im wesentlichen aus dem **Hauptprogramm** und der **Interruptroutine**. Beide sollten von Ihnen geschrieben bzw. ergänzt werden. Die Interrupt-Routine klinkt sich in den Hardware-Interrupt **09** ein, um den Scancode in dem Moment mitzubekommen, wo die Tastatur ihn erzeugt. Sie wird also immer dann aktiv, wenn wir eine Taste drücken, oder loslassen. Das Hauptprogramm wird also an einer unbekannten Stelle unterbrochen. Die Warscheinlichkeit ist allerdings groß, dass das Hauptprogramm gerade im Interrupt **16hex** hängt und auf einen Tastendruck wartet.

Die Interruptroutine ist wie folgt aufgebaut:

- Zunächst werden alle Register auf dem Stapel gesichert, die im folgenden verwendet werden. Da keine großen Aktionen durchgeführt werden, kann man auf das Umstellen des Stapels verzichten, wir verwenden den fremden (unbekannten) Stapel.
- Anschließend sollen die Zeichen **0Dhex** und **0Ahex** ausgegeben werden, um den Cursor an den Anfang der nächsten Bildschirmzeile zu setzen. Hierfür darf **keinesfalls ein DOS-Interrupt** verwendet werden, denn DOS ist nicht *reentrant*. Das bedeutet, dass man aus einem DOS-Interrupt heraus keinen neuen (anderen) **DOS**-Interrupt aufrufen darf, weil sonst keine erfolgreiche Rückkehr mehr gelingt. Da man ja nicht weiß, wo der Interrupt **09** auftrat, ist hier Vorsicht angesagt. Mit **BIOS**-Funktionen geht es aber.
- Dann wird der Zeichencode der eben betätigten Taste vom Tastaturprozessor (an Port **60hex**) abgeholt und in ASCII-Zeichen entsprechend dem Hexcode umgewandelt.
- Die beiden ASCII-Zeichen sollen in der richtigen Reihenfolge ausgegeben werden!
- Anschließemd werden alle gesicherten Register vom Stapel zurückgeholt, und es folgt ein Sprung an die ursprüngliche Adresse des Interrupt **09hex**. Die korrekte Adresse wird hier vom Hauptprogramm aus eingetragen.

[20]Meine Schüler im BKT-Assembler-Kurs finden diese Datei auch im Austauschverzeichniss des Schulservers

Die Interruptroutine arbeiten quasi unabhängig vom Hauptprogramm. Immer dann, wenn der Interrupt **09hex** durch die Betätigung einer Taste ausgelöst wird, wird sie aktiv. Die Zeichenausgabe endet nicht mit mit einem Zeilenumbruch, damit vom Hauptprogramm das erhaltene Zeichen in der gleichen Zeile dahinter ausgegeben werden kann. Im Hauptprogramm steht **kein** Zeilenumbruch, der steht am Anfang der Interruptroutine.

Das Hauptprogramm beinhaltet folgende Schritte:

- Der Interrupt **09hex** wird umgestellt. Dazu muss zuerst die Adresse des alten Interrupt **09hex** ermittelt und abgespeichert werden. Danach wird die Adresse der eigenen Routine in die Interrupttabelle eingetragen. Die Interrupttabelle befindet sich ganz am Anfang des Speichers an der Adresse 0000:0000. Da jede Interruptadresse ein Doppelwort belegt, kann man leicht ausrechnen, an welcher Stelle er steht: Adresse der Interruptadresse = Interruptnummer mal vier. **Aber Vorsicht!** Wird die Adresse nicht in einem **einzigen Befehl** ausgelesen oder eingeschrieben, besteht die Gefahr, dass genau zwischen zwei Befehlen ein Interrupt ausgelöst wird. Dann wäre das System in einem inkonsistenten Zustand. Daher empfiehlt es sich, die **Funktionen 25hex** und **35hex** des **DOS-Funktioneninterrupt 21hex** hierfür zu verwenden. Natürlich kann man auch während des Schreib- bzw. Lesezugriffes mit dem Befehl `CLI` die Interrupts sperren.

- Ein Zeichen wird vom Tastaturtreiber abgeholt. Es ist das Zeichen, das der Tastaturtreiber aus dem Scancode der Taste gemacht hat. Diesen Zeichen wird in den Ausgabetext eingetragen und in ASCII-Zeichen entsprechend dem Hexcode umgewandelt. Auch diese ASCII-Zeichen werden in den Text eingetragen. Anschließend wird der Text mit Hilfe der **DOS-Funktion 09hex** des Interrupt **21hex** ausgegeben. Da ja kurz zuvor eine Taste gedrückt worden sein muss, steht der Ausgabetext immer **hinter dem Scancode** des entsprechenden Tastendrucks.

- Der Vorgang wiederholt sich so lange, bis die Taste <**Esc**> gedrückt wird. Danach wird das Programmende eingeleitet. Vor dem eigentlichen Programmende muss jedoch unbedingt die ursprüngliche Interruptadresse wieder in die Interrupttabelle eingetragen werden. Auch hier sind wieder die erwähnten Vorsichtsmaßnahmen zu beachten. Probieren Sie ruhig einmal aus, ob (oder wann) der Rechner abstürzt, wenn das Zurückspeichern vergessen wurde.

Wenn das Programm fertig ist, testen Sie doch einmal aus, welche Scancodes von welchen Tasten erzeugt werden. Fällt Ihnen etwas auf?

Hier das Listing des unfertigen Programms:

```
; Programm SCANCODE
; Das Programm stellt den Scancode aller Tasten sowie die damit erzeugten
; Zeichen auf dem Bildschirm dar.
jmp  START
;--------
TEXT:       db '    Zeichen: '
ZEICHEN     db 'x Code: '    ; Hier wird das Zeichen eingetragen
ZEICHEN_H   dw 0        ; hier wird der Zeichencode (höherwertig) eingetragen
ZEICHEN_L   dw 0        ; hier wird der Zeichencode (niederwertig) eingetragen
            db 'hex$'
;--------
AUSGABE:     ; Ausgabe des Zeichens in AL mit Funktion 0Eh des Int 10h
  mov  AH,0E ; Das Unterprogramm wird nur von der Interruptroutine verwendet.
  int  010
ret
;-- Prozedur HEXCODE wandelt Zeichen in AL in ASCII-Hexcode in AX
;   Dabei steht das höherwertige Zeichen in AH, das niederwertige in AL
HEXCODE:
  mov  AH,AL  ; Wert auch nach AH kopieren
  and  AL,0F  ; höherwertigen Teil in AL löschen
  shr  AH,4   ; höherwertigen Teil in AH isolieren
  add  AX,03030 ; beide Teile in ASCII-Zeichen wandeln
  cmp  AL,'9' ; ist Zahl bis einschließlich Ziffer 9?
  jbe  >L1    ; wenn ja, fertig, überspringen
  add  AL,7   ; sonst noch Korrekturwert addieren
L1:
  cmp  AH,'9' ; ist Zahl bis einschließlich Ziffer 9?
  jbe  ret    ; wenn ja, fertig
  add  AH,7   ; sonst noch Korrekturwert addieren
ret
;-------
```

```
NEUER_INT09:  ; Zusätzliche vorgeschaltete Interruptroutine für Int. 09h
;-- Register sichern, die in der Routine benötigt werden:

;-- die Zeichen 0Dh und 0Ah für Zeilenende/neue Zeile ausgeben:




;-- Scancode von der Tastatur (aus Port 60h) nach AL holen:

;-- Scancode in Hexcode umwandeln:
;   (Hierzu darf das gleiche Unterprogramm wie das vom Hauptprogramm
;    benutzte verwendet werden.)

;-- Höherwertiges Byte des ASCII-Zeichens ausgeben:




;-- Niederwertiges Byte des ASCII-Zeichens ausgeben:

;-- gesicherte Werte in Register zurückspeichern:

;-- Zur ursprünglichen Interrupt-Routine springen:


;-- Hauptprogramm: -----------------------------------------------
START:
;-- Adresse des alten Interrupt 09h ermitteln:


;-- alte Adresse in Sprungadresse am Ende der Ergänzungsroutine einpatchen:


;-- Adresse des neuen Interrupt 09h in Interrupttabelle eintragen:



S1:                ; Hauptschleifenanfang
;-- Zeichen von der Tastatur holen:
  mov  AH,010
  int  016
;--
  mov  ZEICHEN,AL   ; Zeichen in Text eintragen
```

```
  push AX           ; Zeichencode sichern
    mov  AL,AH      ; höherwertiges Byte nach AL zur Umwandlung
    call HEXCODE    ;   und in ASCII Hexcode umwandeln
    xchg AL,AH      ; Bytes ordnen
    mov  ZEICHEN_H,AX ; und in Text eintragen
  pop  AX           ; Zeichencode zurückholen
  call HEXCODE      ; niederwertiges Byte in ASCII-Hexcode umwandeln
  xchg AL,AH        ; Bytes ordnen
  mov  ZEICHEN_L,AX ;   und in Text eintragen
;-- Ausgabe des Textes:
  mov  DX,TEXT
  mov  AH,09
  int  021
;-- Wiederholen, wenn nicht <Esc> gedrückt wurde:
  cmp  ZEICHEN,01B
  jne  S1           ; Schleife, bis <Esc> gedrückt wird
ENDE:
;-- Interrupt 09 wieder auf ursprünglichen Wert zurückstellen:



;-- Programmende:
  mov  AH,04C
  int  021
```

21 Einbinden von Assembler in Hochsprachen

Meist werden Assemblerroutinen nur erstellt, um Programme in Hochsprachen schneller ablaufen zu lassen. Die Routinen müssen also in Hochsprachen **eingebunden** werden. Am Beispiel von `PASCAL` sollen zwei Varianten vorgestellt werden, wie man das machen kann.

Als erste Möglichkeit besteht in `PASCAL` die Möglichkeit, Assemblercode direkt einzugeben. Dazu verwendet man das Schlüsselwort `ASM`. Alle Befehle in den nachfolgenden Zeilen bis zum nächsten `END` werden dann als Assemblerbefehle eingegeben. Hierbei werden (wie auch sonst unter `PASCAL` üblich) Hex-Werte mit einem **vorangestellten Dollarzeichen** gekenzeichnet. Als Beispiel soll nachfolgende Sequenz dienen, die uns als Textausgabe bekannt vorkommen sollte.

```
asm
  mov  AH,$09
  mov  DX,Offset TEXT
  int  $21
end;
```

Dieses Verfahren eignet sich jedoch nur für relativ kurze Befehlssequenzen.

Die zweite Möglichkeit besteht darin, komplette Funktionen und Prozeduren in Assembler zu schreiben, die dann als **Objektcode** compiliert werden. `PASCAL` kann diese dann **einbinden**.

Mit diesem Verfahren wollen wir uns hier näher beschäftigen. Dazu ist es zunächst notwendig, einige Grundzüge der Hochsprache (hier `PASCAL`) zu kennen.

Von Hochsprachen werden (fast) keine Register verwendet, in denen über mehrere Befehle hinweg irgendwelche Werte verwaltet werden. Die Ausnahme: **BP**. [21] Das Register **BP** wird in **jeder Prozedur und Funktion** verwendet, um **lokale Variablen** zu verwalten. Diese werden beim Eintritt in das Unterprogramm auf dem Stapel angelegt und **BP** dient dabei als Zeiger, um diese zu adressieren. Der Stapelzeiger **SP** wird dazu entsprechend weiter gestellt. In einem in Assembler geschriebenen Unterprogramm muß also unbedingt der Zeiger **BP** beim Verlassen den gleichen Wert wie beim Eintritt in das Unterprogramm haben. Dies gilt nicht für **AX**, **BX**, **CX**, **DX** oder **SI** und **DI**.[22] Auch die Segmentadressen in **DS** und **SS** müssen unbedingt erhalten bleiben.

Die nächste Frage, die zu klären ist, ist die Frage der Art des Aufrufes. Die Antwort hängt davon ab, ob wir ein Unterprogramm für eine **Unit** oder für das **Hauptprogramm**

[21] Im **Protected Mode** betrifft das **EBP** anstelle von **BP**.

[22] Bei der Sprache **C** ist das bezüglich **SI** und **DI** eventuell anders.

schreiben.[23] Ein Unterprogramm im Hauptprogramm wird mit einem `NEAR CALL` aufgerufen, ein Unterprogramm in einer Unit dagegen mit einem `FAR CALL`. Entsprechend muss unsere Routine dann mit einem `RET` (Hauptprogramm) oder mit einem `RETF` (Unit) enden, je nach dem, wofür es compiliert wird.[24]

Vielen Unterprogrammen werden beim Aufruf bestimmte Parameter übergeben. Das geschieht dadurch, dass die entsprechenden Werte in der Reihenfolge, in der sie in `PASCAL` definiert sind (also von links nach rechts), auf den Stapel gebracht werden, bevor das Unterprogramm mit einem `CALL` oder einem `CALL FAR` (siehe oben, Stichwort Unit) aufgerufen wird. Es ist dann die Aufgabe des **Unterprogramms**, diese Daten vom Stapel wieder zu entfernen.[25] Aus diesem Grund enden `PASCAL`-Unterprogramme, denen Werte übergeben wurden, mit einer Zahl zusätzlich zum `RET`- bzw. `RETF`-Befehl. Die Zahl gibt an, um wieviele Einheiten der Stapelzeiger **SP** nach dem Rücksprung wieder erhöht werden muss.

Ein Beispiel:
Die Definition der Procedure `BILDSCHIRM_LOESCHEN` sieht unter `PASCAL` so aus:
`PROCEDURE BILDSCHIRM_LOESCHEN(Loeschzeichen: Char; Farbe: Byte); External;`
Dabei bedeutet `External`, dass die Procedure „extern“, also in einer Object-Datei außerhalb des `PASCAL`-Quellcodes zu finden ist. Die beiden Werte `Loeschzeichen` und `Farbe` haben zwar nur die Länge 1 Byte, da sie aber mit einem `PUSH` auf den Stapel gelangen, belegen sie jeder dennoch ein ganzes Wort. Nach Abschluss der Routine müssen also 2 Worte (entsprechend 4 Byte) vom Stapel genommen werden. Also endet das Unterprogramm mit `RET 4`. Ist es für eine Unit geschrieben, dann heißt der Rücksprung entsprechend `RETF 4`.

Unter `PASCAL` sieht das „Gerüst“ dieses Unterprogramms damit so aus:

```
BILDSCHIRM_LOESCHEN:
 push BP
   mov BP,SP
   ...
   ...
 pop  BP
ret  4
```

Sehen Sie sich die Struktur eines jeden Unterprogramms anhand des vorstehend dargestellten Beispiels an. Als erstes wird immer **BP** gesichert, danach wird **BP** auf den aktuellen Stapeloffset eingestellt. Es folgen die eigentlichen Befehle des Unterprogramms.

[23] Alle Ausführungen gelten nur für die Sprache *PASCAL*, in anderen Hochsprachen kann das anders sein!

[24] Bei Programmiersprachen, die Programme für den **Protected Mode** erzeugen, ist es anders. Da gibt es nur den *FAR CALL*.

[25] Bei **C** ist dies auch wieder anders. Hier werden die Übergabewerte in der Reihenfolge von **rechts nach links** auf den Stapel gelegt und von dem **aufrufenden** Programmteil wieder entfernt.

Am Schluss wird **BP** zurückgeholt und es folgt der Rücksprung in Hauptprogramm mit dem abschließenden „Aufräumen“ des Stapels – hier 2 Worte entsprechend 4 Byte – wie bereits beschrieben. Zur Adressierung der übergebenen Werte findet **BP** Verwendung. BP-Zeiger[26] beziehen sich immer auf das **Stacksegment**.

Damit sind wir auch schon bei der Frage, welche Variablen auf welche Art und Weise übergeben werden. Einfach ist es bei Variablen in **Byte**- und **Wort**-Länge wie `Byte`, `Char`, `Boolean`, `Word` und `Integer`. Diese werden mit `PUSH` auf den Stapel gebracht[27] und belegen auf jeden Fall ein ganzes **Wort**. Bei **Byte**-Variablen ist damit natürlich nur das **niederwertige Byte** gültig, das höherwertige Byte hat einen zufälligen Wert.

Bei einem **Doppelwort** wie `Longint` werden **zwei Worte** auf den Stapel gelegt. Dabei kommt das höherwertige Wort zuerst auf den Stapel, liegt also an der höheren Adresse. **Real**-Variablen sind Dreifachworte, wobei wiederum das höchstwertigste Wort zuerst auf den Stapel kommt. Dann gibt es noch **Zeiger-Variable** wie `Pointer`, `Array`, `String` oder `Record`. Hier kommt nicht die Variable selbst, sondern nur ein **Zeiger auf die Variable** auf den Stapel. Dadurch kann vom Unterprogramm direkt auf die Original-Daten zugegriffen werden. **Bei allen Zeigern wird zuerst die Segmentadresse, dann der Offset auf den Stapel gelegt.** Die Frage, die sich stellt: Wie genau greife ich auf die übergebenen Variablen zu?

Nachdem die Eingangssequenz durchlaufen ist und **BP** und **SP** den gleichen Wert haben, zeigt **BP** immer auf **die Stelle im Stacksegment, wo der alte BP-Wert gesichert wurde**. Ein Wort darüber, also an der Stelle `W[BP+2]` liegt somit die **Rücksprungadresse**. Wenn unsere Assemblerroutine nicht für eine Unit sondern für das Hauptprogramm[28] geschrieben wurde, dann ist der nächste Wert auf dem Stapel bei `[BP+4]` der **zuletzt** von `PASCAL` auf den Stapel gelegte Übergabewert. Schreiben wir für eine Unit, dann belegt die Rücksprungadresse 2 Worte, also finden wir den letzten Übergabeparameter dann bei `[BP+6]`. Bleiben wir aber in unserem Beispiel mit einem `NEAR CALL`. Als letztes Wort wurde wie gesagt ein Byte für die Farbe an der Stelle `[BP+4]` auf den Stapel gelegt, davor das Löschzeichen, mit dem der Bildschirm überschrieben werden soll. Dieses liegt somit bei `B[BP+6]`.

[26]Das gilt im **Protected Mode** entsprechend für **EBP** anstelle von **BP**.

[27]Auch dies gilt nur für *PASCAL*. Es gibt auch Hochsprachen (z.B *DELPHI*, die einige Parameter in Registern übergeben.

[28]In diesem Fall belegt die Rücksprung-Adresse 2 Bytes, bei einer Unit 4 Byte

Schreiben wir eine `FUNCTION`, dann gibt diese einen Wert an das aufrufende Programm-Teil zurück. Dieser Wert wird je nach Größe in einem (oder mehreren) bestimmten Register erwartet. In `PASCAL`[29] werden folgende Register verwendet:

- für Byte-Variablen: AL
- für Wort-Variablen: AX
- für Doppelwort-Variablen: DX:AX
- für Dreifach-Wort-Variablen: DX:BX:AX

Es kann nun sein, dass innerhalb der `Assembler`-Routinen auch auf Variablen zugegriffen werden muss, die im `PASCAL`-Programm definiert sind. Das ist kein Problem. Sie werden einfach am Anfang der `Assebler`-Routine als `EXTRN` deklariert. Dabei muss noch die Größe angegeben werden, damit der Compiler beim Erstellen des Objekt-Codes den entsprechenden Platz frei hält. Schließlich müssen wir noch dem Compiler mitteilen, wie der erzeugte Code eingebunden werden kann. Das geschieht mit der Direktrive `CODE SEGMENT BYTE PUBLIC`[30] im Programmkopf der Assemblerroutine.

Sie finden dazu nachfolgend[31] die beiden Dateien `GROSS.ASM` und `GROSS.PAS` als Beispiel. In der `PASCAL`-Datei `GROSS.PAS` finden Sie in den Zeilen 7 bis 12 die Deklarationen für 3 Prozeduren und 2 Funktionen, die als `Assembler`-Routinen in `GROSS.ASM` geschrieben sind. Vergleichen Sie, in welcher Reihenfolge in der Prozedur `Bildschirm_Loeschen` die übergebenen Variablen deklariert sind (und demnach auf den Stapel kommen) mit der Nummerierung `[BP+x]`, unter der sie von der Assemblerroutine angesprochen werden. Vergleichen Sie auch die Zahl der übergebenen Variablen der jeweiligen Prozeduren und Funktionen mit der Zahl, die jeweils hinter dem `RET` steht. Schaun Sie sich in dem Beispiel an, in welchen Registern bei den Funktionen `GrossBuchstabe` und `Hole_Taste` die Werte zurückgegeben werden.

[29] Das ist natürlich wiederum in jeder Hochsprache anders.
[30] Auch diese Angabe muss in anderen Hochsprachen eventuell anders sein.
[31] Meine Schüler im BKT-Assembler-Kurs finden diese Dateien auch im Austauschverzeichniss des Schulservers

Hier kommt das Listing des PASCAL-Programms `GROSS.PAS`:

```
PROGRAM GROSS; {Demonstration zum Einbinden von Assembler-Routinen in PASCAL}

Var Cursor_X, Cursor_Y: Byte;
    Zeichen: Char;

{Es folgen die externen Funktionen und Prozeduren:}
PROCEDURE Cursor_Setzen; EXTERNAL;
PROCEDURE Bildschirm_Loeschen(Loeschzeichen: Char; Farbe: Byte); EXTERNAL;
FUNCTION GrossBuchstabe(Buchstabe:Char):Char; EXTERNAL;
FUNCTION Hole_Taste:Char; EXTERNAL;
PROCEDURE Schreibe(Buchstabe: Char); EXTERNAL;
{$L GROSS.OBJ}  {Anweisung an Compiler: Hier sind die Prozeduren zu finden.}

BEGIN
  Bildschirm_Loeschen(' ',$1E); {Löschen mit blauem Hintergrund}
  Cursor_X := 10;
  Cursor_Y := 5;
  Cursor_Setzen;
  WriteLn('Alle eingegebenen Zeichen werden in Großbuchstaben gewandelt.');
  Cursor_X := 32;
  Cursor_Y := 8;
  Cursor_Setzen;
  WriteLn('Abbruch mit <Esc>');
  Cursor_X := 1;
  Cursor_Y := 20;
  Cursor_Setzen;
  REPEAT
    Zeichen := GrossBuchstabe(Hole_Taste);
    Schreibe(Zeichen);
  UNTIL Zeichen = #27;
  Bildschirm_Loeschen(' ',$07); {Löschen in Standard-Farbe}
  Cursor_X := 1;
  Cursor_Y := 1;
  Cursor_Setzen;        {Cursor nach oben links}
END.
```

Es folgt das Listing des Assemblerteils `GROSS.ASM`:

```
; Hilfsroutinen für PASCAL-Programm GROSS.PAS
; Compilieren mit dem Befehl:
;
;   A386 GROSS.8 +O
; oder:
;   A386 GROSS.8 GROSS.OBJ
;
;=========================================
CODE SEGMENT BYTE PUBLIC      ; compilieren für PASCAL
EXTRN CURSOR_X:B, CURSOR_Y:B  ; Byte-Variablen, sind unter PASCAL definiert
;=========================================
CURSOR_SETZEN:     ; Cursor auf Position CURSOR_X/CURSOR_Y setzen
  mov  AH,2        ; Funktionsnummer
  mov  BH,0
  mov  DL,CURSOR_X
  mov  DH,CURSOR_Y
  dec  DH          ; Zählweise
  dec  DL          ;    anpassen
  int  010
ret
;=========================================
BILDSCHIRM_LOESCHEN:
; Überschreibt den Bildschirm (25 Zeilen zu 80 Zeichen)
; mit dem übergebenen Löschzeichen in übergebener Farbe.
  push BP              ; BP sichern
    mov  BP,SP         ; Basiszeiger "verankern"
    mov  AX,0B800      ; ES auf Bildschirmsegment
    mov  ES,AX         ;    einstellen
    mov  AL,[BP+6]     ; übergebenes Lösch-Zeichen nach AL einlesen
    mov  AH,[BP+4]     ; übergebene Farbe nach AH einlesen
    mov  CX,80*25      ; Zahl der Bildschirmzeichen
    cld                ; Richtung auf "Vorwärts"
    xor  DI,DI         ; beim ersten Bildschirmzeichen anfangen
    rep  stosw         ; alles überscheiben
  pop BP               ; BP restaurieren
ret 4
;=========================================
```

```
GROSSBUCHSTABE:
; Alle Buchstaben einschließlich aller (nicht nur deutscher) Umlaute
; werden in die zugehörigen Großbuchstaben gewandelt
; Ersatz für UPCASE
  push BP              ; BP sichern
    mov  BP,SP
    mov  AL,[BP+4]     ; übergebene Variable nach AL einlesen
    cmp  AL,'a'        ; Tastencode < 97 ?
    jb   >F1           ; dann fertig
    cmp  AL,'ñ'        ; Tastencode über 164 ?
    ja   >F1           ; dann Ende
    cmp  AL,'z'        ; Tastencode > 122 ?
    ja   >M1           ; dann weitere Untersuchung
    and  AL,11011111xB ; Standard-Kleinbuchstaben wandeln
    jmp  >F1           ; und fertig
M1:
    cmp  AL,084        ; Taste = ä ?
    je   >A1
    cmp  AL,094        ; Taste = ö ?
    je   >O1
    cmp  AL,081        ; Taste = ü ?
    je   >U1
    cmp  AL,082        ; Taste = é ?
    je   >E2
    cmp  AL,086        ; Taste = å ?
    je   >A2
    cmp  AL,087        ; Taste = ç ?
    je   >C1
    cmp  AL,091        ; Taste = æ ?
    je   >E1
    cmp  AL,0A4        ; Taste = ñ ?
    jne  >F1           ; wenn nein, fertig
    inc  AL            ; Taste := Ñ
    jmp  >F1
E1:
    mov  AL,092        ; Taste := Æ
    jmp  >F1
C1:
    mov  AL,080        ; Taste := Ç
    jmp  >F1
A2:
    mov  AL,08F        ; Taste := Å
    jmp  >F1
```

```
E2:
    mov  AL,090         ; Taste := É
    jmp  >F1
U1:
    mov  AL,09A         ; Taste := Ü
    jmp  >F1
A1:
    mov  AL,08E         ; Taste := Ä
    jmp  >F1
O1:
    mov  AL,099         ; Taste := Ö
F1:
  pop BP                ; BP restaurieren
ret 2                   ; Rücksprung, 1 Wort vom Stapel entfernen
;=========================================
HOLE_TASTE:             ; Holt ein Zeichen vom Tastaturpuffer ab
  mov  AH,010
  int  016              ; das Zeichen wird in AL geliefert, das passt
ret
;=========================================
SCHREIBE:               ; Schreibt ein Zeichen im TTY-Modus
  push BP
    mov  BP,SP
    mov  AL,B[BP+4]     ; übergebenes Zeichen laden
  pop  BP               ; (man kann BP auch jetzt schon zurückholen)
  mov  AH,0E            ; Funktionsnummer Zeichenausgabe
  int  010
ret 2                   ; Rücksprung, 1 Wort vom Stapel entfernen
;=========================================
```

22 Übungsaufgaben, Teil 6:

22.1 Aufgabe 6.1

Sie sollen eine Routine schreiben, mit dere Hilfe man von PASCAL aus an einer beliebigen Stelle auf dem Bildschirm einen String in beliebiger Farbe schreiben kann, ohne den Cursor zu verstellen. Zum Testen dieser Routine existiert bereits die Datei ASTEST.PAS, die nachfolgend dargestellt ist.[32] Diese Datei ist bis auf Zeile 11 und 12 fertig. Tragen Sie dort bitte in Zeile 11 die Deklaration der PROCEDURE SCHREIB ein. Dieser Prozedur werden (der Reihe nach) folgende Parameter übergeben:

1. die Spaltennummer auf dem Bildschirm (Byte-Variable)
2. die Zeilennummer auf dem Bildschirm (Byte-Variable)
3. die Farbnummer für die Ausgabe (Byte-Variable)
4. Zeiger auf den auszugebenden String

In Zeile 12 ist dann die Anweisung an den Compiler einzutragen, wie die Objekt-Datei heißt, in der die einzubindenden Routinen enthalten sind. Die zugehörige Assemblerdatei soll ASTEST.ASM heißen. Diese müssen Sie neu erstellen, jedoch können Sie aus GROSS.ASM die bestehenden Prozeduren CURSOR_SETZEN, BILDSCHIRM_LOESCHEN sowie die Funktion HOLE_TASTE unverändert verwenden. Einzig neu ist nur die von Ihnen zu schreibende Prozedur SCHREIB, die einen Text in einer beliebigen Farbe an eine bestimmte Stelle auf dem Bildschirm schreibt. **Denken Sie daran, dass für den String *zwei Worte* auf den Stapel kommen, erst die Segmentadresse und danach die Offsetadresse**.

Hier noch ein Tip: Unter PASCAL steht im **ersten Byte** eines String immer die **Stringlänge**. Eine Stringendemarke gibt es nicht!

Beachten Sie, dass Sie beim Compilieren mit A386 eine Objekt-Datei erzeugen wollen und keine COM-Datei! Es muss daher ein großes **O** mit einem + davor als Parameter für den Assembler mit angegeben werden. Der zugehörige Befehl lautet damit:
A386 ASTEST.ASM +O

Die PASCAL-Datei können Sie mit dem Komandozeilencompiler TPC.EXE kompilieren. Der zugehörige Befehl lautet:
TPC ASTEST.PAS

Anmerkung: Die Procedure CURSOR_SETZEN mit dem direkten Zugriff auf Variable ist nicht sonderlich sinnvoll, sie soll nur zeigen, wie man es machen kann.

[32] Meine Schüler im BKT-Assembler-Kurs finden diese Datei auch im Austauschverzeichniss des Schulservers

Hier folgt das Listing des fast fertigen Testprogramms ASTEST.PAS.

```
PROGRAM ASTEST;  {Demonstration zum Einbinden von Assembler-Routinen}

Var Cursor_X, Cursor_Y: Byte;
    Zeichen: Char;
    i: Byte;

{Es folgen die externen Funktionen und Prozeduren:}
PROCEDURE Cursor_Setzen; EXTERNAL;
  {Setzt Cursor an Stelle CURSOR_X/CURSOR_Y}
PROCEDURE Bildschirm_Loeschen(Loeschzeichen: Char; Farbe: Byte); EXTERNAL;
  {Überschreibt Bildschirm mit LOESCHZEICHEN in der Farbe FARBE}
FUNCTION Hole_Taste:Char; EXTERNAL;  {Holt ein Zeichen von der Tastatur}



Begin
  Bildschirm_Loeschen(' ',$1E); {Löschen mit blauem Hintergrund}
  Schreib(22,5,$4F,'*-----------------------------------*');
  for i:=6 to 15 do Schreib(22,i,$4F,'!                                   !');
  Schreib(22,16,$4F,'*-----------------------------------*');
  Schreib(35,8,$4E,'Schreibtest');
  Schreib(31,12,$4E,'Abbrechen mit <Esc>');
  REPEAT
    Zeichen:=Hole_Taste;
  UNTIL Zeichen = #27;
  Bildschirm_Loeschen(' ',$07); {Löschen in Standard-Farbe}
  Cursor_X := 1;
  Cursor_Y := 1;
  Cursor_Setzen;        {Cursor nach oben links}
END.
```

23 Musterlösungen, Teil 1

23.1 Aufgabe 1.1 (ABFRAGE)

Frage: Warum steht in der ersten Programmzeile (Zeile 7 im Quelltext) kein `Offset` vor `TEXT`?
Antwort: Hier ist der Bezeichner `TEXT` nicht als Variable deklariert worden, sondern als Label. Man erkennt das an dem Doppelpunkt dahinter, der bei Variablen „fehlt". Der Wert eines Labels ist immer seine Adresse. Genau die soll ja nach **DX** geladen werden. Wir haben hier also den Fall, dass eine Variable existiert (nämlich mit dem Ausgabetext als Inhalt), die **keinen** Variablennamen hat.

23.2 Aufgabe 1.2 (ABFRAGE1)

Eine Lösung könnte etwa so aussehen:

```
; Programm ABFRAGE1
; Das Programm gibt die Anleitung aus (Programmende mit Taste q oder Q)
; und beendet sich, sobald diese Taste gedrückt wird. Sonst macht das
; Programm nichts.
;========================== Anfang des Programmcodes ===================
; Zuerst erfolgt eine Textausgabe mit Bedienungsanleitung:
  mov  DX,TEXT   ; Offset-Adresse des Textes für Interrupt laden (s. u.)
  mov  AH,09     ; Funktionsnummer für Textausgabe
  int  021       ; Textausgabe mit DOS-Interrupt ausführen
;-------------
S1:   ; Schleifenanfang - Schleifendurchlauf, so lange nicht 'q' gedrückt wird
; Das nächste Zeichen von der Tastatur abholen:
  mov  AH,010    ; Funktionsnummer "Zeichen lesen" für BIOS-Tastatur-Interrupt
  int  016       ; Ausführen des Interrupt - das Zeichen steht danach in AL
  cmp  AL,'Q'    ; ist das in AL übergebene Zeichen ein "Q"?
  je   >L1       ; wenn ja, springe zum Programmende
  cmp  AL,'q'    ; ist das in AL übergebene Zeichen ein "q"?
  jne  S1        ; wenn nein, Sprung zurück zum Schleifenanfang
;-------------
L1:
; Die Taste "q" oder "Q" wurde gedrückt, das Programmende wird eingeleitet:
  mov  AH,04C    ; Funktionsnummer für Programmende
  int  021       ; Aufforderung an DOS, das Programmende auszuführen
;========================== Ende des Programmcodes =======================
; In dem nachfolgenden Bereich ist der Ausgabetext definiert.
; Der Ausgabestring muss mit einem "$" abgeschlossen sein, so verlangt es
; DOS für seine Service-Routine 09 des Interrupt 21h zur Textausgabe.

TEXT: db 'Programmende mit Taste Q oder q!$'
```

23.3 Aufgabe 1.3 (ABFRAGE2)

Eine Lösung könnte etwa so aussehen:

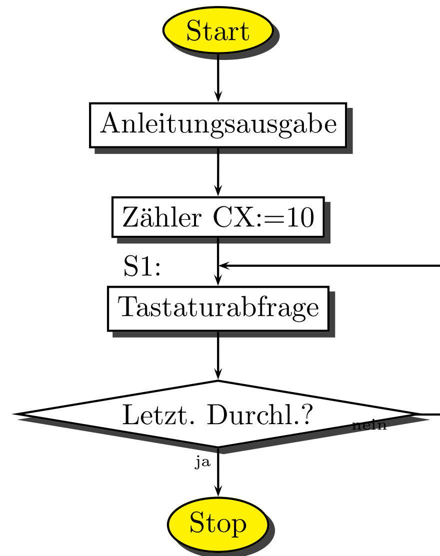
```
; Programm ABFRAGE2
; Das Programm gibt die Anleitung aus (Programmende mit Taste q oder Q)
; und beendet sich, sobald diese Taste gedrückt wird. Sonst macht das
; Programm nichts.
;=========================== Anfang des Programmcodes ===================
; Zuerst erfolgt eine Textausgabe mit Bedienungsanleitung:
  mov  DX,TEXT   ; Offset-Adresse des Textes für Interrupt laden (s. u.)
  mov  AH,09     ; Funktionsnummer für Textausgabe
  int  021       ; Textausgabe mit DOS-Interrupt ausführen
;-------------
S1:   ; Schleifenanfang - Schleifendurchlauf, so lange nicht 'q' gedrückt wird
; Das nächste Zeichen von der Tastatur abholen:
  mov  AH,010    ; Funktionsnummer "Zeichen lesen" für BIOS-Tastatur-Interrupt
  int  016       ; Ausführen des Interrupt - das Zeichen steht danach in AL
;-- Das abgeholte Zeichen wird auf dem Bildschirm dargestellt:
  mov  AH,0E     ; Funktionsnummer für Zeichenausgabe
  int  010       ; Zeichen ausgeben
;-------------
  cmp  AL,'Q'    ; ist das in AL übergebene Zeichen ein "Q"?
  je   >L1       ; wenn ja, springe zum Programmende
  cmp  AL,'q'    ; ist das in AL übergebene Zeichen ein "q"?
  jne  S1        ; wenn nein, Sprung zurück zum Schleifenanfang
;-------------
L1:
; Die Taste "q" oder "Q" wurde gedrückt, das Programmende wird eingeleitet:
  mov  AH,04C    ; Funktionsnummer für Programmende
  int  021       ; Aufforderung an DOS, das Programmende auszuführen
;========================== Ende des Programmcodes ========================
; In dem nachfolgenden Bereich ist der Ausgabetext definiert.
; Der Ausgabestring muss mit einem "$" abgeschlossen sein, so verlangt es
; DOS für seine Service-Routine 09 des Interrupt 21h zur Textausgabe.

TEXT: db 'Programmende mit Taste Q oder q!$'
```

23.4 Aufgabe 1.4 (10TASTEN)

Sehr hilfreich für dieses Programm ist der komfortable Befehl `LOOP`. Er bewirkt folgendes:

- Zunächst wird das (Zähl)-Register **CX** um 1 vermindert.
- Dann wird geprüft, ob **CX=0** ist. Ist das (noch) nicht der Fall, wird zu dem angegebenen Sprungziel gesprungen.

Dadurch eignet sich dieser Befehl besonders für Schleifen, die eine fest vorgegebene Anzahl von Durchläufen machen sollen.



Programm 10TASTEN

Nebenstehend ist das zugehörige Flussdiagramm angegeben. Nach der Anleitungstextausgabe wird das Zählregister **CX** auf 10 (die Anzahl der gewünschten Schleifendurchläufe) gesetzt. Der Schleifenanfang hat die Sprungmarke `S1` bekommen. Innerhalb der Schleife wird lediglich die Tastatur abgefragt (mit dem Interrupt 16hex, Funktion 10hex), denn die **Anzahl** der Tastendrücke bis zum Beenden des Programms soll ja gezählt werden.

Natürlich besteht keine Verpflichtung, den Befehl `LOOP` zu verwenden. Man könnte ihn auch ersetzen durch die Befehlssequenz:

```
dec  CX   ; Zähler herunterzählen (wird CX=0, dann wird CF gesetzt)
jnz  S1   ; Wenn CF nicht gesetzt, Sprung zum Schleifenanfang
```

Eine dritte Alternative bestünde darin, ein Zählregister bei 0 starten zu lassen und am Schluss immer zu prüfen, ob die 10 schon erreicht ist. Das wären dann aber schon 3 Befehle (anstelle eines einzigen):

```
inc  CX    ; Zähler hochzählen
cmp  CX,10 ; Schon bis 10 hochgezählt?
jne  S1    ; Wenn nein, Sprung zum Schleifenanfang
```

Die Frage nach der günstigsten Lösung stellt sich wohl nicht.

Das Listing des Programms mit dem LOOP-Befehl könnte etwa so aussehen:

```
; Programm 10TASTEN
; Das Programm gibt die Bedienungsanleitung aus und beendet sich, sobald die
; 10. Taste gedrückt wird. Sonst macht das Programm nichts.
;=========================== Anfang des Programmcodes =====================
; Zuerst erfolgt eine Textausgabe mit Bedienungsanleitung:
  mov  DX,TEXT   ; Offset-Adresse des Textes für Interrupt laden (siehe unten)
  mov  AH,09     ; Funktionsnummer für Textausgabe
  int  021       ; Textausgabe mit DOS-Interrupt ausführen
;-------------
; Die Tastendrücke werden von Zählschleifenbefehl LOOP in CX gezählt
  mov  CX,10     ; Zähl-Register auf Startwert 10 setzen.
;-------------
S1:   ; Schleifenanfang - Schleifendurchlauf, so lange nicht 'q' gedrückt wird
; Das nächste Zeichen von der Tastatur abholen:
  mov  AH,010    ; Funktionsnummer "Zeichen lesen" für BIOS-Tastatur-Interrupt
  int  016       ; Ausführen des Interrupt - das Zeichen steht danach in AL
;-------------
  loop S1        ; CX runterzählen. Wenn noch nicht 0 zum Schleifenanfang
;-------------
; Die 10. Taste wurde gedrückt, das Programmende wird eingeleitet:
  mov  AH,04C    ; Funktionsnummer für Programmende
  int  021       ; Aufforderung an DOS, das Programmende auszuführen
;=========================== Ende des Programmcodes =======================
; In dem nachfolgenden Bereich ist der Ausgabetext definiert.
; Der Ausgabestring muss mit einem "$" abgeschlossen sein, so verlangt es
; DOS für seine Service-Routine 09 des Interrupt 21h zur Textausgabe.

TEXT: db 'Programmende nach 10 Tastendrücken!$'
```

Möglicherweise wollten Sie nicht den LOOP-Befehl verwenden. Dann wäre beispielsweise auch folgende Version möglich:

```
; Programm 10TASTEN
; Das Programm gibt die Bedienungsanleitung aus und beendet sich, sobald die
; 10. Taste gedrückt wird. Sonst macht das Programm nichts.
;========================== Anfang des Programmcodes ====================
; Zuerst erfolgt eine Textausgabe mit Bedienungsanleitung:
  mov  DX,TEXT   ; Offset-Adresse des Textes für Interrupt laden (siehe unten)
  mov  AH,09     ; Funktionsnummer für Textausgabe
  int  021       ; Textausgabe mit DOS-Interrupt ausführen
;-------------
; Die Tastendrücke möchte ich im Register CL zählen.
  mov  CL,0      ; Zähl-Register auf Startwert 0 setzen.
;-------------
S1:   ; Schleifenanfang - Schleifendurchlauf, so lange nicht 'q' gedrückt wird
; Das nächste Zeichen von der Tastatur abholen:
  mov  AH,010    ; Funktionsnummer "Zeichen lesen" für BIOS-Tastatur-Interrupt
  int  016       ; Ausführen des Interrupt - das Zeichen steht danach in AL
;-------------
  inc  CL        ; Zählerstand der Tastendrücke um 1 erhöhen
  cmp  CL,10     ; ist der Zählerstand auf 10 angestiegen?
  jb   S1        ; so lange noch kleiner, weiter in Schleife
;-------------
; Die 10. Taste wurde gedrückt, das Programmende wird eingeleitet:
  mov  AH,04C    ; Funktionsnummer für Programmende
  int  021       ; Aufforderung an DOS, das Programmende auszuführen
;========================== Ende des Programmcodes =======================
; In dem nachfolgenden Bereich ist der Ausgabetext definiert.
; Der Ausgabestring muss mit einem "$" abgeschlossen sein, so verlangt es
; DOS für seine Service-Routine 09 des Interrupt 21h zur Textausgabe.

TEXT: db 'Programmende nach 10 Tastendrücken!$'
```

Hierbei wird in **CL** hochgezählt, bis 10 erreicht ist. Wie man leicht erkennt, ist aber die Verwendung des `LOOP`-Befehles eleganter.

23.5 Aufgabe 1.5 (ZIFFER)

Nebenstehend ist ein Flussdiagramm für das Programm `ZIFFER` dargestellt. Die Funktionsweise stellt sich wie folgt dar:

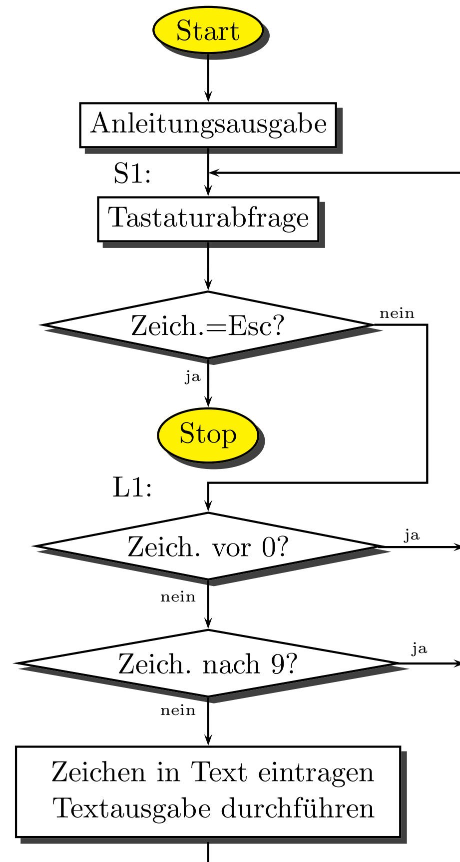
Zuerst gibt das Programm eine Anleitung aus. Danach beginnt die Schleife des Programms, markiert mit der Sprungmarke `S1`.

Zum Anfang der Schleife wird die Tastatur abgefragt. Danach wird geprüft, ob die Taste <**Esc**> gedrückt wurde. Ist das der Fall, beendet sich das Programm. Anderenfalls wir zur Sprungmarke `L1` gesprungen.

Für einen Hochsprachen-Programmierer mag es überraschen, dass das Programmende mitten im Hauptprogramm stehen kann. Unter `ASSEMBLER` ist jedoch (fast) alles möglich.

Nun wird geprüft, ob der Zeichencode kleiner war, als der der ASCII-Null. Wenn ja, muss nichts ausgegeben werden, es erfolgt ein Sprung zum Schleifenanfang.

Danach wird geprüft, ob der Zeichencode größer als der der 9 war. Auch in diesem Fall erfolgt ein Sprung zurück zum Schleifenanfang.



Programm ZIFFER

Wurden beide Fragen verneint, dann wurde eine Zifferntaste gedrückt. Das Zeichen wird nun in den Ausgabestring eingefügt, der zugehörige Text wird ausgegeben. Danach erfolgt der Sprung zurück zum Schleifenanfang.

Es folgt auf der nächsten Seite die Lösung, die zum dargestellten Flussdiagramm passt.

```
; Programm ZIFFER
; Das Programm gibt die Bedienungsanleitung aus (ATEXT) und beendet sich,
; sobald die Taste <Esc> gedrückt wird.
;=========================== Anfang des Programmcodes =====================
; Zuerst erfolgt eine Textausgabe mit Bedienungsanleitung:
  mov  DX,ATEXT  ; Offset-Adresse des Textes für Interrupt laden (siehe unten)
  mov  AH,09     ; Funktionsnummer für Textausgabe
  int  021       ; Textausgabe mit DOS-Interrupt ausführen
;-------------
S1:   ; Schleifenanfang - Durchlauf, so lange nicht <Esc> gedrückt wird
; Das nächste Zeichen von der Tastatur abholen:
  mov  AH,010    ; Funktionsnummer "Zeichen lesen" für BIOS-Tastatur-Interrupt
  int  016       ; Ausführen des Interrupt - das Zeichen steht danach in AL
;-------------
  cmp  AL,01B    ; ist das in AL übergebene Zeichen ein <Esc>?
  jne  >L1       ; wenn nein, Sprung zum Weitermachen
  mov  AH,04C    ; Funktionsnummer für Programmende
  int  021       ; Aufforderung an DOS, das Programmende auszuführen
;-------------
L1:
; Wir prüfen, ob das Zeichen eine Ziffer ist.
  cmp  AL,'0'    ; Ist das Zeichen kleiner, als der Wert der ASCII-0?
  jb   S1        ; wenn vor 0, Sprung zum Schleifenanfang
  cmp  AL,'9'    ; Ist das Zeichen größer, als der Wert der ASCII-9?
  ja   S1        ; wenn nach 9, Sprung zum Schleifenanfang
;-------------
; Das Zeichen ist eine Ziffer und muss in den Ausgabestring eingefügt werden.
  mov  ZIFFER,AL ; Das Zeichen in den Ausgabestring (ZTEXT) eintragen
  mov  DX,ZTEXT  ; Offset-Adresse des Textes für Interrupt laden
  mov  AH,09     ; Funktionsnummer für Textausgabe
  int  021       ; Textausgabe mit DOS-Interrupt ausführen
  jmp  S1        ; und weiter gehts am Schleifenanfang
;========================== Ende des Programmcodes =======================
; In dem nachfolgenden Bereich sind die Ausgabetexte definiert.

ATEXT: db 'Anzeige von Zifferntasten, Programmende mit Taste <Esc>.$'
ZTEXT: db 0D,0A,'Dies ist die Taste '
ZIFFER db '0.$'
```

Wer bisher nur Hochsprachen kennen gelernt hat, für den wird es sicher gewöhnungsbedürftig sein, dass sich ein Text über mehrere unterschiedlich definierte Bereiche erstrecken kann. Da der Textteil bei `ZTEXT:` nicht mit einem $ abgeschlossen ist, hört die Ausgabe hier nicht auf. Der Restteil, der als Variable `ZIFFER` deklariert wurde, wird auch noch mit ausgegeben.

24 Musterlösungen, Teil 2:

24.1 Aufgabe 2.1 (ZEICHEN)

An der Stelle im Program, wo `TEXT2` durch einen `INT 021` ausgegeben wird (Zeile 12), verändert sich im Register **AL** der Wert. Das eingegebene Zeichen ist dann weg, es befindet sich anschließend eine 24h in AL. Eine Prüfung, ob <**Esc**> gedrückt wurde, geht dann ins Leere.

Abhilfe kann ein `PUSH AX` **vor** und ein `POP AX` **nach** der Textausgabe schaffen. Das Programm sähe dann so aus:

```
; Programm ZEICHEN.ASM
jmp  START; Sprung über den Bereich der Variablen
;---- Ab hier Deklaration von Variablen ----
TEXT1: db 'Bitte eine Taste drücken, Abbruch mit <Esc>',0A,0D,'$'
TEXT2: db 0A,0D,'Sie haben die Taste '
BUCHSTABE db 'x'
  db ' gedrückt.$' ; (Resttext von TEXT2)
;---- Ende der Variablendeklaration ------
START:
  mov DX,TEXT1 ; Adresse des Anleitungstextes nach DX
  mov AH,09    ; Funktionsnummer für Stringausgabe
  int 021      ; Ausgabe des Anleitungstextes Text1
S1:
;-- Ein Zeichen wird von der Tastatur geholt:
  mov AH,010
  int 016
;--
  mov BUCHSTABE,AL ; Das Zeichen wird in den Ausgabetext "eingeflickt"
;-- Es folgt die Ausgabe des Textes:
  push AX      ; Zeichen auf Stapel sichern
    mov DX,TEXT2
    mov AH,09
    int 021
  pop AX       ; gesichertes Zeichen nach AL zurückholen
;--
  cmp AL,01B   ; war das Zeichen das <Esc>?
  jne S1       ; wenn nein, weiter in Schleife
  mov AH,04C   ; sonst: Programmende einleiten
  int 021
```

24.2 Aufgabe 2.2 (TASTEN)

Nebenstehend ist ein Flussdiagramm zum Programm `TASTEN` dargestellt. Hier soll zunächst die Funktionsweise erläutert werden.

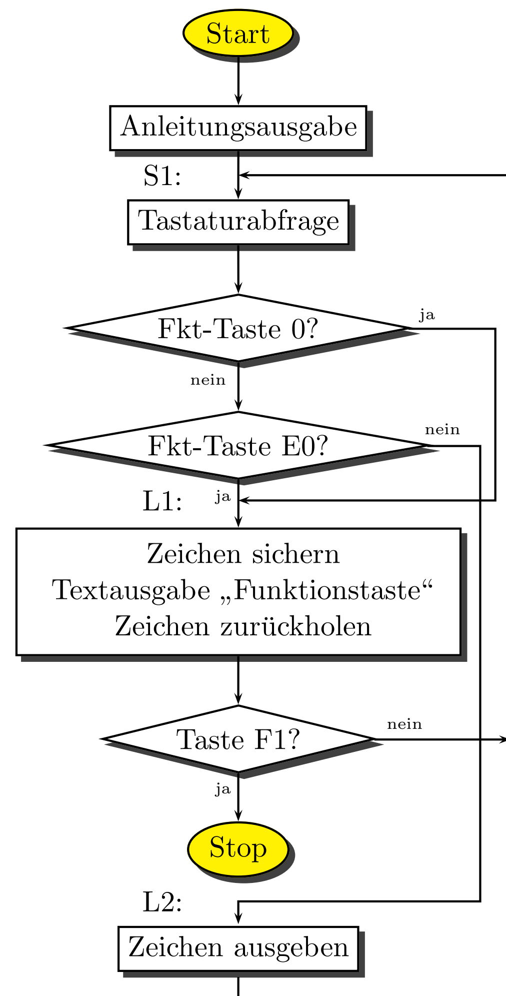
Zuerst gibt das Programm eine Anleitung aus. Nach dem Programmteil zur Ausgabe befindet sich der Schleifenanfang des Programms.

Zum Schleifenbeginn wird die Tastatur abgefragt.

Sobald ein Zeichen vorliegt, wird es daraufhin geprüft, ob es eine Funktionstaste vom Typ **0** ist (also ob der Inhalt von **AL** Null ist). Ist das der Fall, erfolgt ein Sprung zur Sprungmarke `L1`.

Anderenfalls erfolgt eine Prüfung, ob eine Funktionstaste vom Typ **E0** gedrückt wurde. Ist das nicht der Fall, erfolgt ein Sprung zur Sprungmarke `L2`.

Anderenfalls gelangt man zur Sprungmarke `L1`, wo man auch hingekommen wäre, wenn eine Funktionstaste vom Typ **0** gedrückt hätte. Es erfolgt die Textausgabe
`Sie haben eine Funktionstaste gedrückt.`
Zuvor muss jedoch das Zeichen auf dem Stapel gesichert werden. Anschließend wird es zurückgeholt.



Programm ZIFFER

Nun wird geprüft, ob die Funktionstaste <**F1**> gedrückt wurde. Wenn ja, beendet sich das Programm. Anderenfalls erfolgt ein Sprung zurück zum Schleifenanfang.

Wurde zur Sprungmarke `L2` gesprungen, dann wurde **keine** Funktionstaste betätigt. Daher erfolgt jetzt nur noch die Ausgabe eines einzelnen Zeichens, nämlich des Zeichens, das zur gedrückten Taste gehört. Danach erfolgt ein Sprung zurück an den Schleifenanfang.

Das Listing des zu diesem Flussdiagramm gehörenden Programms ist auf der nächsten Seite dargestellt.

```
; Programm TASTEN
;=================================================================
  jmp  START    ; Sprung über den Bereich der Variablen
;----  Ab hier Deklaration von Variablen  ----------
TEXT1:  db 'Bitte eine Taste drücken, Abbruch mit <F1>',0D,0A,'$'
TEXT2:  db 0D,0A,'Sie haben eine Funktionstaste gedrückt.$'
;----  Ende des Variablenbereiches  ----------------
START:          ; Hier beginnt das eigentliche Programm
  mov  DX,TEXT1 ; Adresse des Anleitungstextes nach DX laden
  mov  AH,09    ; Funktionsnummer für Stringausgabe
  int  021      ; Ausgabe des Anleitungstextes
;----  Anfang der Schleife
S1:
;--  Ein Zeichen wird von der Tastatur geholt:
  mov  AH,010   ; Funktionsnummer zum Holen eines Zeichens
  int  016      ; Tastatur-Interrupt
;--  Das abgeholte Zeichen befindet sich jetzt in AL.
  cmp  AL,0     ; Ist es eine Funktionstaste (Typ 0)?
  je   FUNKTIONSTASTE ; wenn ja, Sprung
  cmp  AL,0E0   ; Ist es eine Funktionstaste (Typ E0)?
  jne  NORMALE_TASTE  ; wenn nein, Sprung
FUNKTIONSTASTE:
;--  Eine Funktionstaste wurde gedrückt.
  push AX       ; Zeichen auf dem Stapel sichern
;--  Es folgt die Textausgabe:
    mov  DX,TEXT2
    mov  AH,09
    int  021    ; Ausgabe des Textes mit Funktion 09 des DOS-Int 21h
;--
  pop  AX       ; gesichertes Zeichen zurückholen
  cmp  AH,03B   ; Wurde die Taste <F1> gedrückt?
  jne  S1       ; wenn nein, weiter in Schleife
  mov  AH,04C   ; sonst: Programmende einleiten
  int  021      ; Programm beenden
;-----------------------------------------------------
NORMALE_TASTE:
;-- Eine normale Taste wurde gedrückt. Das Zeichen muss ausgegeben werden.
  mov  AH,0E
  int  010      ; Ausgabe des Zeichens über BIOS-Interrupt
  jmp  S1       ; weiter gehts beim Schleifenanfang
```

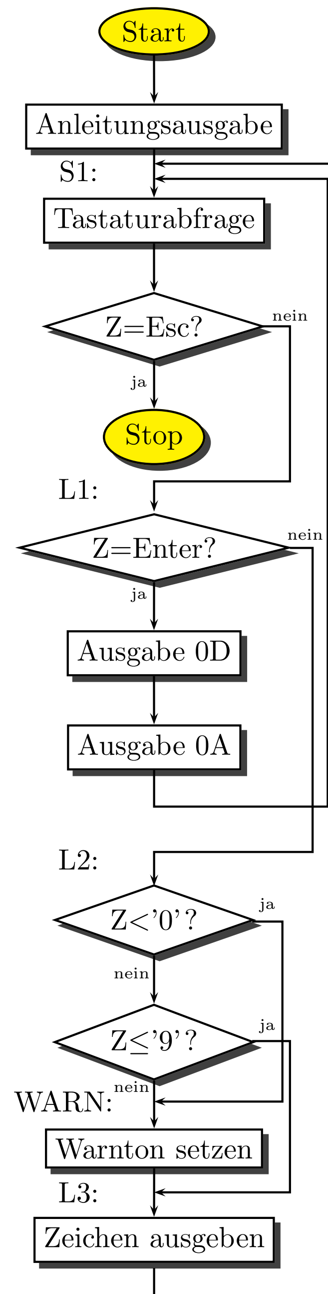
25 Musterlösungen, Teil 3:

25.1 Aufgabe 3.1 (ZIFFERN)

Nebenstehend ist ein mögliches Flussdiagramm zur Lösung dargestellt. Natürlich ist das nicht die einzige mögliche Lösung. Auf der nächsten Seite folgt dann das zugehörige Programm im Quellcode. Hier einige Anmerkungen zur Struktur des Programms.

Nach der Ausgabe des Anleitungstextes wird die Tastatur abgefragt. Wird ein Zeichen eingegeben, wird zunächst geprüft, ob es ein <**Esc**> war. Wenn ja, beendet sich das Programm, anderenfalls wird das Programmende übersprungen. Es mag den einen oder anderen überraschen, dass das Programmende mitten im Flussdiagramm steht. Das ist unter Assembler ohne weiteres möglich. Anschließend wird geprüft, ob <**Enter**> gedrückt wurde. War das der Fall, werden nacheinander die beiden Zeichen 0D und 0A für den Zeilenumbruch ausgegeben und das Programm springt zum Schleifenanfang zur nächsten Tastaturabfrage.

Wurde nicht <**Enter**> gedrückt, springt das Programm weiter zu `L2`. Dort wird das Zeichen geprüft, ob es eine Ziffer ist. Das geschieht zweistufig. Zunächst wird geprüft, ob der Wert des ASCII-Zeichens kleiner als '0'(=30h) ist. Ist das der Fall, ist es **keine** Ziffer, es erfolgt ein Sprung zu `WARN`. Anderenfalls wird geprüft, ob der Wert des ASCII-Zeichens kleiner oder gleich als '9'(=30h) ist. Ist das der Fall, haben wir eine Ziffer und es erfolgt ein Sprung nach `L3` zur Zeichenausgabe. Anderenfalls landen wir bei `WARN`, wo das Zeichen gegen eine 07 ausgetauscht wird. Die 07 ist das Steuerzeichen für einen Piepton, der dann bei der anschließenden Zeichenausgabe ertönt. Danach geht es zurück zum Schleifenanfang, wo wieder die Tastatur abgefragt wird.

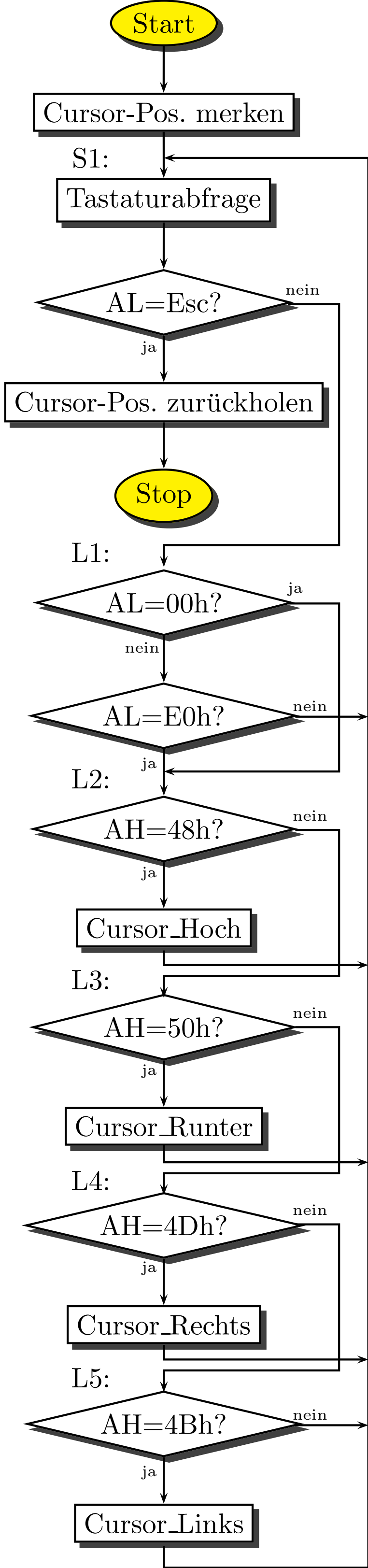


Programm ZIFFERN

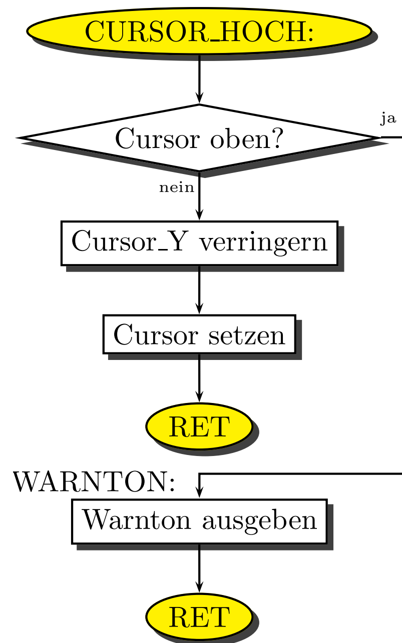
Die einspaltige gestreckte Struktur des Flussdiagramms ermöglicht eine 1:1-Umsetzung in den Quelltext. Dieser ist nachfolgend dargestellt.

```
; Programm ZIFFERN
;========================== Anfang des Programmcodes ====================
; Zuerst erfolgt eine Textausgabe mit Bedienungsanleitung:
  mov  DX,ATEXT ; Offset-Adresse des Textes für Interrupt laden (siehe unten)
  mov  AH,09    ; Funktionsnummer für Textausgabe
  int  021      ; Textausgabe mit DOS-Interrupt ausführen
;-------------
S1:; Schleifenanfang - Schleifendurchlauf, so lange nicht <Esc> gedrückt wird
; Das nächste Zeichen von der Tastatur abholen:
  mov  AH,010   ; Funktionsnummer "Zeichen lesen" für BIOS-Tastatur-Interrupt
  int  016      ; Ausführen des Interrupt - das Zeichen steht danach in AL
;-------------
  cmp  AL,01B   ; ist das in AL übergebene Zeichen ein <Esc>?
  jne  >L1      ; wenn nein, Programmende überspringen
  mov  AH,04C   ; Funktionsnummer für Programmende
  int  021      ; Aufforderung an DOS, das Programmende auszuführen
;==================================================================
L1:             ; Wir prüfen, ob <Enter> gedrückt wurde:
  cmp  AL,0D
  jne  >L2      ; wenn nicht <Enter>, dann weiterspringen
  call AUSGABE  ; Ausgabe des Zeichens 0D
  mov  AL,0A
  call AUSGABE  ; Ausgabe des Zeichens 0A
  jmp  S1       ; weiter bei Schleifenanfang
;-------------
L2:             ; Wir prüfen, ob das Zeichen eine Ziffer ist.
  cmp  AL,'0'   ; Ist das Zeichen kleiner, als der Wert der ASCII-0?
  jb   WARN     ; wenn vor 0, Sprung zur Warntonausgabe
  cmp  AL,'9'   ; Ist das Zeichen größer, als der Wert der ASCII-9?
  jbe  >L3      ; wenn bis 9, Sprung zur Zeichenausgabe
;-------------
WARN:           ; Ein Warnton wird ausgegeben
  mov  AL,07    ; einen Warnton anfordern
L3:             ; Das Zeichen in AL muss ausgegeben werden:
  call AUSGABE
  jmp  S1       ; und weiter gehts am Schleifenanfang
;============= Unterprogramm: ====================================
AUSGABE:        ; Ausgabe eines Zeichens, das in AL übergeben wird
  mov  AH,0E
  int  010
ret
;============== Ausgabetext: =====================================
ATEXT: db 'Bitte Ziffern eingeben, Programmende mit Taste <Esc>. $'
```

25.2 Aufgabe 3.2 (CURSOR)



Vorstehend ist das Flussdiagramm des Hauptprogramms dargestellt. Zunächst wird die aktuelle Cursorposition gespeichert. Nach Abfrage der Tastatur wird geprüft, ob <**Esc**> gedrückt wurde. Wenn ja, wird der Corsor zurückgestellt und das Programm beendet. Anderenfalls erfolgt eine Prüfung auf `00` oder `E0` (Funktionstasten wie Cursor o.ä.). Wenn keine Funktionstaste vorliegt, erfolgt ein Rücksprung zum Schleifenanfang. Wenn doch, wird der Reihe nach abgeprüft, ob eine der vier Cursortasten gedrückt wurde. Die Codierung der Cursortasten sind aus den Kommentaren im Listing des Programms ersichtlich. Der Cursor wird entsprechend bewegt, falls das möglich ist, danach gehts zum Schleifenanfang zur Tastaturabfrage. Ein neues Zeichen kann eingegeben werden.



Funktion CURSOR_HOCH

Dabei sind die Funktionsblöcke `CURSOR_HOCH`, `CURSOR_RUNTER`, `CURSOR_RECHTS` und `CURSOR_LINKS` als **Unterprogramme** ausgeführt. Da diese alle sehr ähnlich aufgebaut sind, wird nebenstehend als Beispiel nur das Flussdiagramm der Funktion `CURSOR_HOCH` dargestellt.

Hierbei kann das Sprungziel `WARNTON` von **allen** Unterprogrammen aus angesprungen werden. Diese Routine, die natürlich mit einem `RET` endet, wird damit von mehreren Unterprogrammen gemeinsam genutzt. Unter Assembler ist so etwas möglich.

Es gibt noch eine weitere Besonderheit, die in dieser Form nur unter Assembler möglich ist. In `CURSOR_HOCH`, `CURSOR_RUNTER` und `CURSOR_RECHTS` lautet jeweils der letzte Befehl vor dem `RET`: `CALL CURSOR_SETZEN`. Im Unterprogramm `CURSOR_LINKS` scheint dieser Befehl vergessen worden zu sein. Da aber auch am Ende der Befehl `RET` fehlt, läuft dieses Unterprogramm sozusagen „von allein“ in die Routine `CURSOR_SETZEN` hinein.

Alle weiteren Detail werden (hoffentlich) aus dem nachfolgenden kommentierten Listing des Beispielprogramms deutlich.

Hier folgt das Listing einer möglichen Lösung des Programms `CURSOR.ASM`:

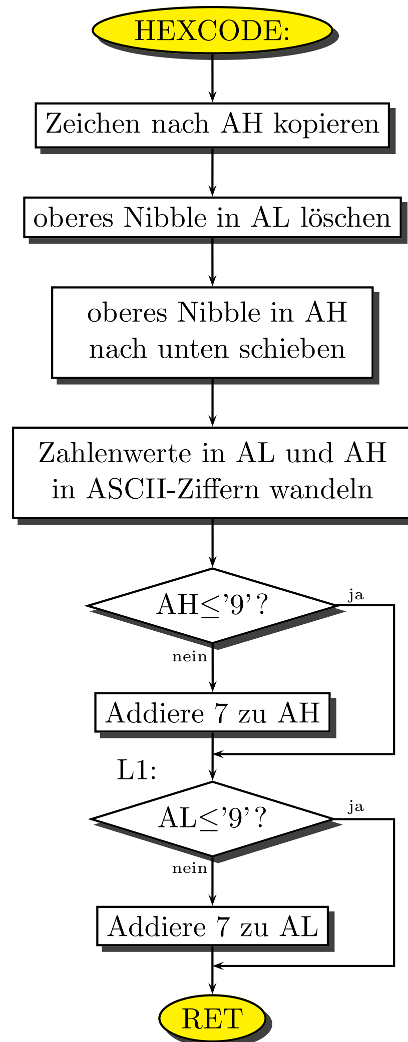
```
; Programm CURSOR.ASM
;==========================================================================
  jmp  START    ; Sprung zum eigentlichen Programm-Anfang
;--------------------------------------------------------------------------
;      Bereich für Variable:
;--------------------------------------------------------------------------
ALTER_CURSOR    dw 0 ; Position des Cursors beim Programmaufruf
BILDSCHIRMSEITE db 0 ; Standard-Bildschirmseite
CURSOR_X        db 0 ; aktuelle x-Position des Cursors
CURSOR_Y        db 0 ; aktuelle y-Position des Cursors
;==========================================================================
;      Diverse Unterprogramme:
;--------------------------------------------------------------------------
WARNTON:        ; Ausgabe eines Warntones
  mov  AX,0E07  ; AH:=Funktionsnummer, AL:=Zeichen für Warnton
  int  010      ; Ausgabe über BIOS
ret
;---------------------------------------
CURSOR_RECHTS:    ; Den Cursor nach rechts bewegen
  cmp  CURSOR_X,79; Steht der Cursor schon in der letzten Spalte?
  je   WARNTON    ; Wenn ja, Warnton ausgeben
  inc  CURSOR_X   ; ansonsten die Spaltennummer erhöhen
  call CURSOR_SETZEN  ; Cursor auf eingestellte Werte setzen
ret
;---------------------------------------
CURSOR_HOCH:      ; Den Cursor nach oben bewegen
  cmp  CURSOR_Y,0 ; Steht der Cursor schon in der obersten Zeile?
  je   WARNTON    ; Wenn ja, Warnton ausgeben
  dec  CURSOR_Y   ; ansonsten die Zeilennummer verringern
  call CURSOR_SETZEN  ; Cursor auf eingestellte Werte setzen
ret
;---------------------------------------
CURSOR_RUNTER:    ; Den Cursor nach unten bewegen
  cmp  CURSOR_Y,24; Steht der Cursor schon in der untersten Zeile?
  je   WARNTON    ; Wenn ja, Warnton ausgeben
  inc  CURSOR_Y   ; ansonsten die Zeilennummer erhöhen
  call CURSOR_SETZEN  ; Cursor auf eingestellte Werte setzen
ret
;---------------------------------------
```

```
CURSOR_LINKS:     ; Den Cursor nach links bewegen
  cmp  CURSOR_X,0 ; Steht der Cursor schon in der ersten Spalte?
  je   WARNTON    ; Wenn ja, Warnton ausgeben
  dec  CURSOR_X   ; ansonsten die Spaltennummer verringern
;---------------------------------------
CURSOR_SETZEN:    ; Setzt den Cursor auf die Stelle CURSOR_X und CURSOR_Y
  mov  DX,W[CURSOR_X] ; DL:=CURSOR_X und DH:=CURSOR_Y
  mov  BH,BILDSCHIRMSEITE
  mov  AH,2       ; Funktionsnummer für "Cursor setzen"
  int  010        ; Ausführen der BIOS-Funktion
ret
;=======================================================================
START:            ;  Anfang des Hauptprogramms
;-- Lesen und Abspeichern der alten Cursorposition:
  mov  AH,03      ; Funktionsnummer: "Cursor lesen"
  mov  BH,BILDSCHIRMSEITE
  int  010        ; BIOS-Funktion "Cursor lesen" ausführen
  mov  ALTER_CURSOR,DX
  mov  W[CURSOR_X],DX ; Beide Cursorwerte gleichzeitig(!) abspeichern
;---
S1:               ; Schleife, bis <Esc> gedrückt wird
  mov  AH,010     ; Ein Zeichen von der Tastatur holen
  int  016
  cmp  AL,01B     ; Taste <Esc> gedrückt?
  jne  >L1        ; wenn nein, überspringen
;-- <Esc> wurde gedrück, Programmende muss eingeleitet werden:
  mov  AX,ALTER_CURSOR  ; Alte Cursorwerte zurückholen
  mov  W[CURSOR_X],AX   ;   und in Cursor_X und Cursor_Y zurückspeichern
  call CURSOR_SETZEN    ; Cursor auf diese Stelle setzen
  mov  AH,04C           ; Funktionsnummer für Programmende
  int  021
;-------------
L1:
  cmp  AL,0     ; Wurde eine Funktionstaste (Typ 0) gedrückt?
  je   >L2      ; wenn ja, überspringen
  cmp  AL,0E0   ; Wurde eine Funktionstaste (Typ E0) gedrückt?
  jne  S1       ; wenn nein, Schleife, nächstes Zeichen holen
L2:
  cmp  AH,048   ; wurde Taste "Cursor-hoch" gedrückt?
  jne  >L3      ; wenn nein, überspringen
  call CURSOR_HOCH ; Cursor nach oben stellen, falls möglich
  jmp  S1       ; zum Schleifenanfang
;-------------
```

```
L3:
  cmp  AH,050   ; wurde Taste "Cursor-runter" gedrückt?
  jne  >L4      ; wenn nein, überspringen
  call CURSOR_RUNTER ; Cursor nach unten stellen, falls möglich
  jmp  S1       ; zum Schleifenanfang
;-------------
L4:
  cmp  AH,04D   ; wurde Taste "Cursor-rechts" gedrückt?
  jne  >L5      ; wenn nein, überspringen
  call CURSOR_RECHTS ; Cursor nach rechts stellen, falls möglich
  jmp  S1       ; zum Schleifenanfang
;-------------
L5:
  cmp  AH,04B   ; wurde Taste "Cursor-links" gedrückt?
  jne  S1       ; wenn nein, zum Schleifenanfang
  call CURSOR_LINKS ; Cursor nach links stellen, falls möglich
  jmp  S1       ; zum Schleifenanfang
;------------------------------------------------------------
```

25.3 Aufgabe 3.3a (HTASTEN)

Das Flussdiagramm des Hauptprogramms ist recht einfach. Daher wird es hier **nicht** gezeigt. Etwas kniffliger gestaltet sich das Unterprogramm `HEXCODE`, das zu einem in **AL** übergebenes ASCII-Zeichen den Zeichencode in hexadezimaler Form ermittelt und die zugehörigen ASCII-Ziffern in **AH** (höherwertig) und **AL** (niederwertig) zurückliefert. Das Flussdiagramm dieses Unterprogrammes ist nebenstehend dargestellt.



Funktion HEXCODE

Als erstes wird das in **AL** übergebene Zeichen auch ins Register **AH** kopiert. So kann in **AH** die **höherwertige** Hex-Ziffer und in **AL** die **niederwertige** Hex-Ziffer erzeugt werden. Dazu wird in **AL** das obere Nibble gelöscht und in **AH** das obere Nibble nach unten geschoben. Anschließend enthalten beide Register den **Zahlenwert**, der der jeweiligen Hex-Ziffer entspricht. Diese müssen jetzt in die ASCII-Zeichen der Zahlen 0 bis 9 bzw. A bis F umgewandelt werden. Für die Zahlen von 0 bis 9 braucht dazu nur eine 3 ins jeweils obere Nibble geschrieben zu werden. Wendet man einen entsprechenden Befehl auf das Register **AX** an, dann kann dies für **AL** und **AH** gleichzeitig geschehen.

Im Prinzip wäre man damit fertig. Tritt als Hex-Ziffer jedoch ein Wert zwischen A und F auf, dann muss der so erzeugte Wert noch korrigiert werden, denn diese Buchstaben schließen sich in der ASCII-Tabelle nicht gleich an die 9 an. Es liegen noch 7 Zeichen dazwischen. Daher müssen **AH** und **AL** nacheinander geprüft werden, ob Werte über 9 vorliegen. Falls ja, muss dann noch jeweils 7 addiert werden.

Hier folgt das Listing einer möglichen Lösung des Programms HTASTEN:

```
; Programm HTASTEN.ASM
; Das Programm gibt zu jedem Zeichen den Hexcode aus.
  jmp  START  ; Sprung zum Anfang des Hauptprogramms
;--------  Variablenbereich:  ----------
ANLEITUNG:
  db 'Bitte eine Taste drücken!',0D,0A,'Beenden mit <ESC>',0D,0A,'$'
AUSGABETEXT:
  db 'Sie haben die Taste '
ZEICHEN db 'x gedrückt, Zeichencode '
ZEICHENCODE DW 0
  db ' hex.',0D,0A,'$'
;---------  Unterprogramme:  ---------
TEXTAUSGABE: ; Gibt Text aus. Adresse in DX übergeben:
  mov  AH,09    ; Funktionsnummer Textausgabe
  int  021      ; Durchführen
ret
;------------------
TASTATUREINGABE:
  mov  AH,010   ; Funktionsnummer Tastaturabfrage
  int  016      ; Durchführen
ret
;------------------
HEXCODE: ; Wandelt Zeichen, das in AL übergeben wird, in Hexcode um.
         ; Rückgabe niederwertig in AL und höherwertig in AH
  mov  AH,AL    ; Zeichen nach AH kopieren
  and  AL,0F    ; oberes Nibble in AL löschen
  shr  AH,4     ; oberes Nibble in AH nach unten schieben
  or   AX,03030 ; Zahlenwerte in AL und AH in ASCII-Ziffern wandeln
  cmp  AH,'9'   ; höherwertige Ziffer '0' bis '9'?
  jbe  >L1      ; wenn ja, überspringen
  add  AH,7     ; Korrektur für 'A' bis 'F'
L1:
  cmp  AL,'9'   ; niederwertige Ziffer '0' bis '9'?
  jbe  ret      ; wenn ja, fertig
  add  AL,7     ; Korrektur für 'A' bis 'F'
ret
;===================
```

```
START: ; Start des Hauptprogramms
  mov  DX,ANLEITUNG
  call TEXTAUSGABE     ; Ausgabe Anleitungstext
S1:                    ; Schleifenanfang
  call TASTATUREINGABE ; Tastaturabfrage
  mov  ZEICHEN,AL      ; Zeichen in Text einbauen
  call HEXCODE         ; Zeichen in ASCII-Ziffern umwandeln
  xchg AL,AH           ; Reihenfolge tauschen
  mov  ZEICHENCODE,AX  ; ASCII-Ziffern in Text einbauen
  mov  DX,AUSGABETEXT  ; Adresse des Ergebnistextes setzen
  call TEXTAUSGABE     ; Ergebnistext ausgeben
  cmp  ZEICHEN,01B     ; wurde <Esc> gedrückt?
  jne  S1              ; wenn nein, weiter in Schleife
  mov  AH,04C          ; Funktionsnummer für Programmende
  int  021             ; Programmende durchführen
```

25.4 Aufgabe 3.3b (HTASTEN2)

Der Aufbau des ergänzten Programms `HTASTEN2` ist nahezu identisch mit dem ursprünglichen Programm `HTASTEN`. Es sind jedoch jetzt **zwei** Worte für die Hex-Ziffern im Ausgabestring vorgesehen (`ZEICHENCODE1` und `ZEICHENCODE2`) und das Unterprogramm `HEXCODE` läuft zwei mal ab, einmal mit dem Zeichen aus **AL** und einmal mit dem Zeichen aus **AH**. Ein neues Flussdiagramm erübrigt sich. Hier das Listing:

```
; Programm HTASTEN2.ASM
; Das Programm gibt zu jedem Zeichen den Hexcode mit Tastaturcode aus.
  jmp  START  ; Sprung zum Anfang des Hauptprogramms
;--------  Variablenbereich:  ----------
ANLEITUNG:
  db 'Bitte eine Taste drücken!',0D,0A,'Beenden mit <ESC>',0D,0A,'$'
AUSGABETEXT:
  db 'Sie haben die Taste '
ZEICHEN db 'x gedrückt, Tastaturcode '
ZEICHENCODE1 DW 0
ZEICHENCODE2 DW 0
  db ' hex.',0D,0A,'$'
;---------  Unterprogramme:  ---------
TEXTAUSGABE: ; Gibt Text aus. Adresse in DX übergeben:
  mov  AH,09    ; Funktionsnummer Textausgabe
  int  021      ; Durchführen
ret
;------------------
TASTATUREINGABE:
  mov  AH,010   ; Funktionsnummer Tastaturabfrage
  int  016      ; Durchführen
ret
;------------------
HEXCODE: ; Wandelt Zeichen, das in AL übergeben wird, in Hexcode um.
         ; Rückgabe niederwertig in AL und höherwertig in AH
  mov  AH,AL    ; Zeichen nach AH kopieren
  and  AL,0F    ; oberes Nibble in AL löschen
  shr  AH,4     ; oberes Nibble in AH nach unten schieben
  or   AX,03030 ; Zahlenwerte in AL und AH in ASCII-Ziffern wandeln
  cmp  AH,'9'   ; höherwertige Ziffer '0' bis '9'?
  jbe  >L1      ; wenn ja, überspringen
  add  AH,7     ; Korrektur für 'A' bis 'F'
L1:
  cmp  AL,'9'   ; niederwertige Ziffer '0' bis '9'?
  jbe  ret      ; wenn ja, fertig
  add  AL,7     ; Korrektur für 'A' bis 'F'
ret
```

```
;===================
START: ; Start des Hauptprogramms
  mov  DX,ANLEITUNG
  call TEXTAUSGABE     ; Ausgabe Anleitungstext
S1:                    ; Schleifenanfang
  call TASTATUREINGABE ; Tastaturabfrage
  mov  ZEICHEN,AL      ; Zeichen in Text einbauen
  push AX              ; erweiterten Tastaturcode sichern
    call HEXCODE         ; Zeichen in ASCII-Ziffern umwandeln
    xchg AL,AH           ; Reihenfolge tauschen
    mov  ZEICHENCODE2,AX ; ASCII-Ziffern in Text einbauen
  pop  AX              ; erweiterten Tastaturcode zurückholen
  mov  AL,AH           ; Tastaturcode nach AL (zur Umwandlung)
  call HEXCODE         ; Zeichen in ASCII-Ziffern umwandeln
  xchg AL,AH           ; Reihenfolge tauschen
  mov  ZEICHENCODE1,AX ; Tastaturcode-Zeichen in Text einbauen
  mov  DX,AUSGABETEXT  ; Adresse des Ergebnistextes setzen
  call TEXTAUSGABE     ; Ergebnistext ausgeben
  cmp  ZEICHEN,01B     ; wurde <Esc> gedrückt?
  jne  S1              ; wenn nein, weiter in Schleife
  mov  AH,04C          ; Funktionsnummer für Programmende
  int  021             ; Programmende durchführen
```

26 Musterlösungen, Teil 4:

26.1 Aufgabe 4.1a (BLINKEIN)

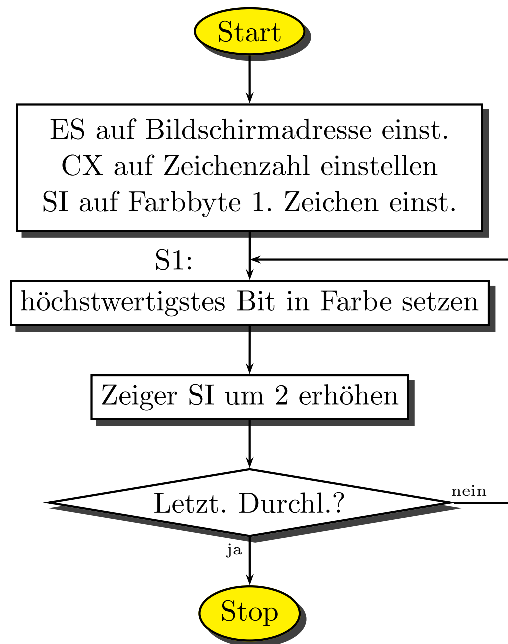
Die Struktur des Programms ist recht einfach, wie das nebenstehende Flussdiagramm zeigt.

In einer Zählschleife wird von jedem Bildschirmzeichen das höchstwertige Bit des Farbbyte gesetzt, bis alle Bildschirmzeichen abgearbeitet sind. Dies geschieht mit dem `OR`-Befehl verbunden mit einem Segment-Override zu **ES**.

Vor dem Einstieg in die Schleife müssen:

- das Segmentregister **ES** auf den Bildschirmspeicher eingestellt werden (Adresse B800hex)
- der Zeiger **SI** auf das Farbbyte des ersten Zeichens eingestellt werden (Adresse 1)
- der Zähler **CX** auf die Anzahl der Bildschirmzeichen (die Anzahl der Schleifendurchläufe) gesetzt werden

Das Programm, dass sich aus diesem Flussdiagramm ergibt, sieht dann wie folgt aus:



Programm BLINKEIN

```
; Programm BLINKEIN
; Das Programm schaltet das Blinken aller Bildschirmzeichen ein.
  mov  AX,0B800  ; Bildschirmadresse ins
  mov  ES,AX     ;   Extrasegment laden
  mov  CX,25*80  ; Startwert für Zählschleife (Zeichenzahl auf Schirm)
  mov  SI,1      ; Zeiger auf Farbbyte des ersten Zeichens
S1:              ; Anfang der Zählschleife
  ES:or B[SI],10000000xB ; höchstwertiges Bit in Farbbyte setzen
  inc  SI        ; Zeiger 2 Byte weiterstellen
  inc  SI
  loop S1        ; Zählschleifenende
  mov  AX,04C00  ; Funktionsnummer Programmende
  int  021
```

Anmerkung 1: Wenn ich keine Lust habe, selbst auszurechnen, wieviele Schleifendurchläufe notwendig sind, um alle Zeichen des Bildschirms zu bearbeiten, kann ich das auch den Assembler machen lassen. Im Befehl `mov CX,25*80` in der dritten Programmzeile geschieht das. Der Compiler rechnet für uns aus, wieviel $25 \cdot 80$ ist und setzt diesen Wert dann ein.

Anmerkung 2: Wie bereits erwähnt, kann es sein, dass je nach Einstellung der Grafikkarte das Blinken nicht funktioniert. Es erscheint dann nur eine andere Hintergrundfarbe. Möglicherweise funktioniert es auch nur im Vollbildmodus. An der Farbänderung kann man dann aber erkennen, dass es korrekt programmiert wurde.

26.2 Aufgabe 4.1b (BLINKAUS)

Analog zu `BLINKEIN.ASM` sieht das Programm `BLINKAUS.ASM` aus. Lediglich der erste Befehl in der Schleife unterscheidet sich, da jetzt das höchstwertige Bit gelöscht wird. Nachstehend folgt das Listing.

```
; Programm BLINKAUS
; Das Programm schaltet das Blinken aller Bildschirmzeichen aus.
  mov  AX,0B800  ; Bildschirmadresse ins
  mov  ES,AX     ;   Extrasegment laden
  mov  CX,25*80  ; Startwert für Zählschleife (Zeichenzahl auf Schirm)
  mov  SI,1      ; Zeiger auf Farbbyte des ersten Zeichens
S1:              ; Anfang der Zählschleife
  ES:and B[SI],01111111xB ; höchstwertiges Bit in Farbbyte löschen
  inc  SI        ; Zeiger 2 Byte weiterstellen
  inc  SI
  loop S1        ; Zählschleifenende
  mov  AX,04C00  ; Funktionsnummer Programmende
  int  021
```

26.3 Aufgabe 4.2 (HALLO)

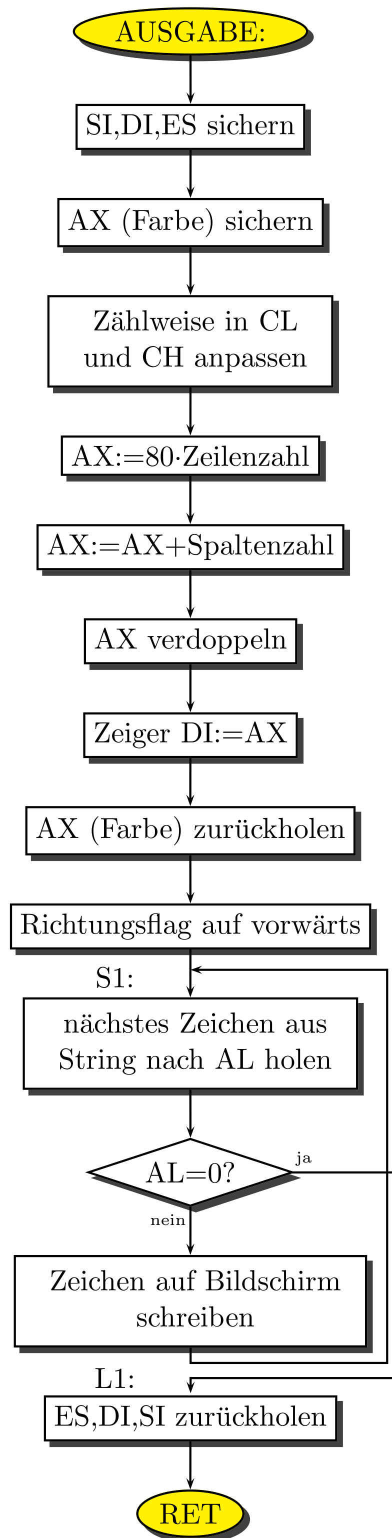
Das Hauptprogramm ist simpel, es enthält keine Verzweigungen. Zunächst werden die Register **AL** und **AH** mit Löschzeichen und Löschfarbe belegt. Dann wird das Unterprogramm `BILDSCHIRM_LOESCHEN` aufgerufen, der Bildschirm wird gelöscht. Danach werden die entsprechenden Register gesetzt, damit das Unterprogramm `AUSGABE` den String ausgeben kann. Zum Schluss erfolgt noch eine Tastaturabfrage, damit das Programm bis zu einem Tastendruck wartet, bevor es sich beendet.

Ähnlich einfach aufgebaut ist auch das Unterprogramm `BILDSCHIRM_LOESCHEN`. Der eigentliche Clou dabei ist der Befehl `rep stosw`. Der Zusatz `rep` bewirkt, dass der Befehl, der dahinter steht, mehrfach ausgeführt wird. Bei jeder Ausführung wird **CX** heruntergezählt, bis CX= 0 ist.

Interessanter ist da schon das Unterprogramm `AUSGABE`. Das zugehörige Flussdiagramm ist nachfolgend dargestellt. Von ein paar kleinen Details abgesehen wird zunächst die Offset-Adresse im Speicher berechnet, ab der der String auf den Bildschirm geschrieben werden soll. **SI** wird als Lesezeiger verwendet, **DI** als Schreibzeiger. In der Schleife ab `S1:` wird immer ein Zeichen aus dem String geholt, dann wird geprüft, ob es die Stringendemarke war, und wenn nicht, wird das Zeichen (zusammen mit dem Farb-Byte) auf den Bildschirm geschrieben. Wenn das **Richtungsflag** zuvor auf **vorwärts** eingestellt wurde, dann bewirken die Befehle `lodsb` und `stosw`, dass die Zeiger dabei sofort weitergestellt werden.

Da dieses Unterprogramm vielfältig verwendbar ist, wurde es so geschrieben, dass Register, die nicht zur Übergabe irgendwelcher Werte verwendet werden, nach Beendigung des Unterprogramms unverändert erhalten bleiben. Die `PUSH`- und `POP`-Befehle am Anfang und am Ende sollen dafür sorgen.

Auf der übernächsten Seite folgt dann das Listing des Programms.



Funktion AUSGABE

```
; Programm HALLO gibt einen String aus und wartet auf Tastendruck.
; Alle Ausgaben über direkten Zugriff auf Bildschirmspeicher
jmp  START
;-- Beginn Variablendeklaration ---------------
TEXT:  db 'Hallo Welt!',0
;--- Ende Variablendeklaration ----------------
;--- Es folgen Unterprogramme  ----------------
BILDSCHIRM_LOESCHEN:   ; Löscht den Bildschirm mit 25 Zeilen und 80 Spalten
                       ; Löschzeichen in AL und Farbe in AH übergeben!
                       ; Alle Registerinhalte bleiben erhalten.
  push CX
  push DI
  push ES
    mov  CX,0B800 ; Bildschirmadresse
    mov  ES,CX    ;   nach ES bringen
    cld           ; Schreibrichtung vorwärts
    mov  CX,25*80 ; Anzahl der zu löschenden Zeichen
    xor  DI,DI    ; Zeiger auf Startwert im Bildschirmspeicher
    rep  stosw    ; Löschen ausführen
  pop  ES
  pop  DI
  pop  CX
ret
;----------------------------------------
AUSGABE: ; Ausgabe eines mit 00h abgeschlossenen String
         ; Übergeben: SI=Adresse des String
         ; AH=Farbe
         ; CL=Bildschirmspalte
         ; CH=Bildschirmzeile
  push SI
  push DI
  push ES
    push AX         ; Farbe zwischendurch sichern, denn AX wird benötigt
      dec  CL       ; Anpassung: 1. Spalte = Spalte 0
      dec  CH       ; Anpassung: 1. Zeile = Zeile 0
      mov  AL,80    ; Zeichenzahl je Zeile
      mul  CH       ; jetzt steht in AX die Zeichenzahl aller Vorzeilen
      mov  CH,0     ; löschen wegen nachfolgender Operation mit CX statt CL!
      add  AX,CX    ; Summe aller vorangehenden Zeichen in AX
      shl  AX,1     ; Multiplikation mit 2
      mov  DI,AX    ; Schreibzeiger auf 1. Bildschirmzeichen
      mov  AX,0B800 ; Bildschirmadresse
      mov  ES,AX    ;   nach ES bringen
    pop  AX         ; Farbe in AH zurückholen
```

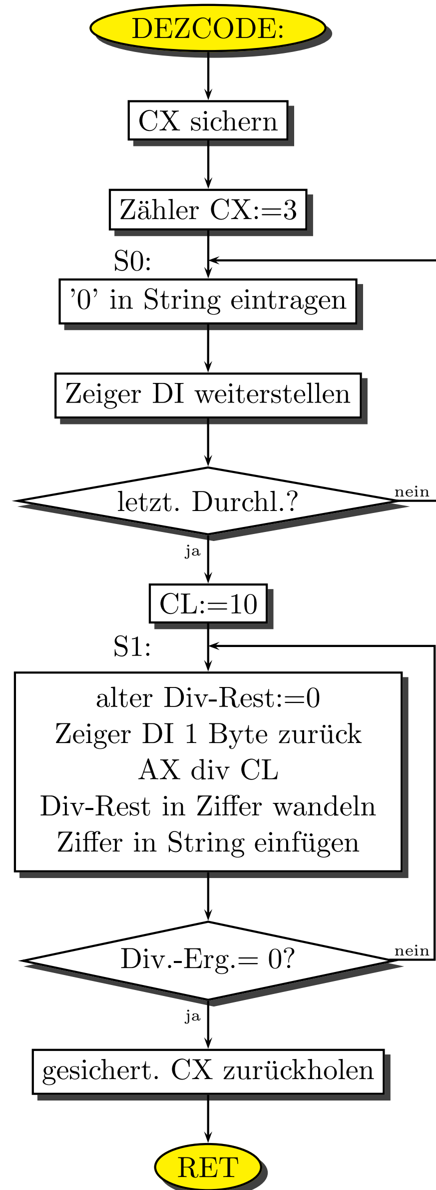
```
    cld             ; Richtungsflag auf VORWÄRTS
S1:
    lodsb           ; nächstes Zeichen nach AL holen
    cmp  AL,0       ; ist es die Ende-Markierung?
    je   >L1        ; wenn ja, fertig
    stosw           ; Zeichen schreiben, einschließlich Farbe!
    jmp  S1         ; Sprung zum Schleifenanfang
L1:
  pop  ES
  pop  DI
  pop  SI
ret
;----------------------------------------
TASTATURABFRAGE:  ; Holt ein Zeichen von der Tastatur
  mov  AH,010     ; Funktionsnummer für "Zeichen holen"
  int  016        ; Tastatur-Interrupt
ret
;========================================
;  HAUPTPROGRAMM
;----------------------------------------
START:
;-- Löschen des Bildschirmes:
  mov  AL,' '     ; Löschzeichen
  mov  AH,01E     ; Lösch-Farbe blau
  call BILDSCHIRM_LOESCHEN ; Löschen mit "Löschzeichen" und "Farbe"
;-- String-Ausgabe:
  mov  SI,TEXT    ; Stringadresse
  mov  CL,1       ; Spalte 1
  mov  CH,10      ; Zeile 10
  mov  AH,01E     ; Farbe hell gelb auf blau
  call AUSGABE    ; Stringausgabe
;--
  call TASTATURABFRAGE ; Auf Tastendruck warten
;-- Programmende:
  mov  AX,04C00
  int  021
```

26.4 Aufgabe 4.3a (DTASTE)

Das Hauptprogramm ähnelt sehr stark dem Programm `HTASTEN`, das Sie bereits geschrieben haben. Daher erübrigt sich hier ein Flussdiagramm; es wäre nahezu identisch. Das interessante an diesem Programm ist die Funktion `DEZCODE`. Hierin wird ein Wert, der in **AL** übergeben wird, in einen **String** umgewandelt, der den Wert als dreistellige Dezimalzahl angibt.

Das Flussdiagramm der Funktion `DEZCODE` ist nebenstehend dargestellt. Sie arbeitet folgendermaßen:

Zunächst werden die drei Ziffern in dem String mit Nullen überschrieben. Dazu wird das für Zählschleifen vorgesehene Zählregister **CX** auf 3 eingestellt (für 3 Schleifendurchläufe). **DI** zeigt zunächst auf den Stringanfang. In der Schleife wird jeweils das ASCII-Zeichen für die Null (30hex) an die Zeigerposition geschrieben, dann wird der Zeiger weitergestellt. Nach Ablauf der Schleife zeigt der Zeiger **DI** auf das Zeichen hinter der letzten Null. Jetzt beginnt die zweite Schleife, in der der Zahlenwert in **AL** immer wieder durch 10 geteilt wird, bis nichts mehr übrig ist. Der Divisionsrest ist dann jeweils die Zahl, die – in eine ASCII-Ziffer umgewandelt – der Reihe nach von hinten nach vorn in den Ergebnis-String eingetragen wird, der die zugehörige Dezimalzahl angibt. Vor Eintritt in die Schleife muss freilich noch ein beliebiges Byte-Register mit dem Teiler 10 belegt werden, da der `DIV`-Befehl nicht mit Konstanten arbeiten kann. Ich habe dafür willkürlich **CL** verwendet.



Funktion DEZCODE

Eine Kleinigkeit muss in der Schleife noch beachtet werden. Der Befehl `DIV CL` bewirkt, dass der Inhalt von **AX** durch den Inhalt von **CL** geteilt wird. Das Ergebnis steht danach in **AL**, der Divisionsrest in **AH**. Daher ist es notwendig, vor jedem Aufruf des `DIV`-Befehles das Register **AH** mit dem Wert Null zu überschreiben. Wir wollen ja eigentlich nur den Inhalt von **AL** durch 10 teilen. Wenn dann in **AH** etwas stände, erhielten wir ein falsches Ergebnis.

Hier das Listing des Lösungsbeispiels:

```
; Programm DTASTE
; Liest Tastencode und stellt ihn auf dem Bildschirm als Dezimal-Zahl dar
jmp  ANFANG
;-- Ab hier die Variablen:
ANLEITUNG: db 'Bitte eine beliebige Taste drücken, Abbruch mit <Esc>',0D,0A,'$'
TEXT:      db 'ASCII-Zeichen: '
BUCHSTABE  db 'x  Zeichencode = '
ZEICHEN:   dd 0 ; hier wird der Zeichencode als ASCII-String eingetragen
                ; Mit "dd" werden 4 Byte belegt. Drei brauchen wir für eine
                ; Zahl zwischen 0 und 255, das vierte Byte (=0) ist die
                ; Stringendemarke für die Textausgabe.
;-- Ende des Variablenbereiches
;==============================================================
;-- Prozedur DEZCODE wandelt Zeichen in AL in ASCII-String.
;   Der Zeiger auf den Stringanfang wird in DI übergeben.
DEZCODE:
  push CX
    mov  CX,3     ; Zahl der zu löschenden Zeichen
S0:
    mov  B[DI],'0'; ein Zeichen mit '0' überschreiben
    inc  DI       ; Zeiger weiter
    loop S0       ; Zählschleife, 3 Durchläufe
;--
    mov  CL,10    ; CL als Teiler (durch 10) festlegen
S1:
    mov  AH,0     ; AX auf AL-Wert setzen
    dec  DI       ; Zeiger 1 zurück
    div  CL       ; Wir teilen AX durch 10. Ergebnis in AL, der Rest in AH
    add  AH,'0'   ; "Rest" in Ziffer wandeln
    mov  B[DI],AH ; Ziffer in String einfügen
    cmp  AL,0     ; Ist das Divisionsergebnis = 0 ?
    jne  S1       ; wenn nein, weiter in Schleife, nächste Ziffer, sonst fertig
  pop  CX
ret
;-----------------------------------------------------
```

```
AUSGABE: ; Ausgabe eines mit 00h abgeschlossenen String
         ; Übergeben: SI=Adresse des String
         ; AH=Farbe
         ; CL=Bildschirmspalte
         ; CH=Bildschirmzeile
  push SI
  push DI
  push ES
    push AX          ; Farbe zwischendurch sichern, denn AX wird benötigt
      dec  CL        ; Anpassung: 1. Spalte = Spalte 0
      dec  CH        ; Anpassung: 1. Zeile = Zeile 0
      mov  AL,80     ; Zeichenzahl je Zeile
      mul  CH        ; jetzt steht in AX die Zeichenzahl aller Vorzeilen
      mov  CH,0      ; löschen wegen nachfolgender Operation mit CX statt CL!
      add  AX,CX     ; Summe aller Vorzeichen in AX
      shl  AX,1      ; Multiplikation mit 2
      mov  DI,AX     ; Schreibzeiger auf 1. Bildschirmzeichen
      mov  AX,0B800 ; Bildschirmadresse
      mov  ES,AX     ;   nach ES bringen
    pop  AX          ; Farbe in AH zurückholen
    cld              ; Richtungsflag auf VORWÄRTS
S1:
    lodsb            ; nächstes Zeichen nach AL holen
    cmp  AL,0        ; ist es die Ende-Markierung?
    je   >L1         ; wenn ja, fertig
    stosw            ; Zeichen schreiben, einschließlich Farbe!
    jmp  S1          ; Sprung zum Schleifenanfang
L1:
  pop  ES
  pop  DI
  pop  SI
ret
;-- Hier beginnt das Hauptprogramm --
ANFANG:
  mov  DX,ANLEITUNG
  mov  AH,09
  int  021             ; Ausgabe des Anleitungstextes
S1:
  mov  AH,010
  int  016             ; Tastatur abfragen
  mov  BUCHSTABE,AL    ; übergebenes Zeichen in Ausgabestring eintragen
  mov  DI,ZEICHEN      ; Zeiger auf Stringanfang für ASCI-Werte der Zahl
  call DEZCODE         ; Zeichen in AL in Ziffernstring wandeln
```

```
;--  Textausgabe mit Farbe:
  mov  SI,TEXT        ; Stringadresse
  mov  CL,1           ; Spalte
  mov  CH,25          ; Zeile
  mov  AH,0E          ; Farbe
  call AUSGABE
  cmp  BUCHSTABE,01B  ; Wurde <Esc> gedrückt? (Programmende)
  jne  S1             ; wenn nein, noch einmal von vorn
;-- Programmende:
  mov  AH,04C
  int  021
```

26.5 Aufgabe 4.3b (DTASTE2)

Da der Unterschied nicht groß ist, kommt hier nur das Listing. Zu beachten ist nur, dass hier das Unterprogramm DEZCODE den umzuwandelden nicht in **AL**, sondern in **AX** übergeben bekommt. Daher muss auch beim Dividieren in diesem Unterprogramm der Teiler 10 in ein Word-Register geladen werden. Entsprechend wird auch nicht (nur) der Inhalt von **AX**, sondern aus **DX:AX** (also ein Doppelregister) beim Dividieren verwendet. Das Ergebnis steht danach in **AX**, der Divisionsrest in **DX**. Logisch ist wohl auch, dass der Ergebnis-String nun nicht nur 3, sondern 5 Stellen hat, da ja Zahlen bis 65535 dargestelt werden können müssen.

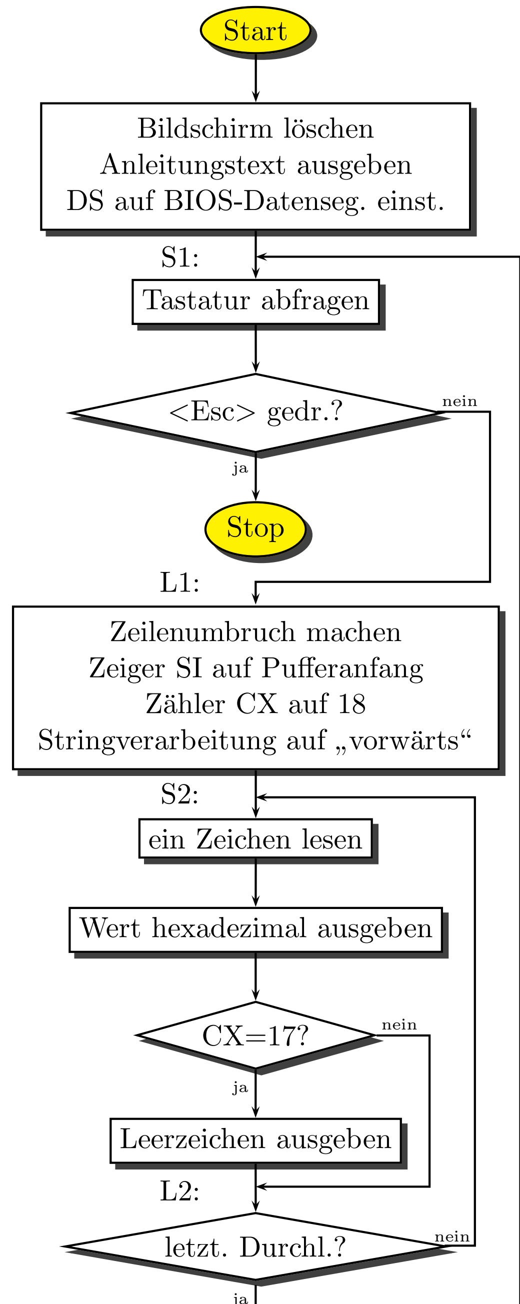
```
; Programm DTASTE2
; Liest Tastencode und stellt ihn auf dem Bildschirm als Hex-Zahl dar
jmp  ANFANG
;-- Ab hier die Variablen:
ANLEITUNG:
  db 'Bitte eine beliebige Taste drücken, Abbruch mit <Esc>',0D,0A,'$'
TEXT:      db 'ASCII-Zeichen: '
BUCHSTABE  db 'x  Zeichencode = '
ZEICHEN:   db 6 dup 0 ; hier wird der Zeichencode als ASCII-String eingetragen
;-- Ende des Variablenbereiches
;================================================
;-- Prozedur DEZCODE wandelt Zeichen in AX in ASCII-Dezimalcode in String
;-- Zeiger auf Stringanfang wird in DI übergeben
DEZCODE:
  push CX
  push DX
    mov  CX,5     ; Zahl der zu löschenden Zeichen
S0:
    mov  B[DI],'0'; ein Zeichen mit '0' überschreiben
    inc  DI       ; Zeiger weiter
    loop S0       ; Zählschleife, 5 Durchläufe
;--
    mov  CX,10    ; CX als Teiler (durch 10) festlegen
S1:
    mov  DX,0     ; AX auf AL-Wert setzen
    dec  DI       ; Zeiger 1 zurück
    div  CX       ; Wir teilen durch 10, der Rest steht in DX
    add  DL,'0'   ; "Rest" in Ziffer wandeln
    mov  B[DI],DL ; Ziffer in String einfügen
    cmp  AX,0     ; Ist das Divisionsergebnis = 0 ?
    jne  S1       ; wenn nein, weiter in Schleife, nächste Ziffer, sonst fertig
  pop  DX
  pop  CX
ret
```

```
;---------------------------------------------------
AUSGABE: ; Ausgabe eines mit 00h abgeschlossenen String
         ; Übergeben: SI=Adresse des String
         ; AH=Farbe
         ; CL=Bildschirmspalte
         ; CH=Bildschirmzeile
  push SI
  push DI
  push ES
    push AX          ; Farbe zwischendurch sichern, denn AX wird benötigt
      dec  CL        ; Anpassung: 1. Spalte = Spalte 0
      dec  CH        ; Anpassung: 1. Zeile = Zeile 0
      mov  AL,80     ; Zeichenzahl je Zeile
      mul  CH        ; jetzt steht in AX die Zeichenzahl aller Vorzeilen
      mov  CH,0      ; löschen wegen nachfolgender Operation mit CX statt CL!
      add  AX,CX     ; Summe aller Vorzeichen in AX
      shl  AX,1      ; Multiplikation mit 2
      mov  DI,AX     ; Schreibzeiger auf 1. Bildschirmzeichen
      mov  AX,0B800 ; Bildschirmadresse
      mov  ES,AX     ;   nach ES bringen
    pop  AX          ; Farbe in AH zurückholen
    cld              ; Richtungsflag auf VORWÄRTS
S1:
    lodsb            ; nächstes Zeichen nach AL holen
    cmp  AL,0        ; ist es die Ende-Markierung?
    je   >L1         ; wenn ja, fertig
    stosw            ; Zeichen schreiben, einschließlich Farbe!
    jmp  S1          ; Sprung zum Schleifenanfang
L1:
  pop  ES
  pop  DI
  pop  SI
ret
```

```
;-- Hier beginnt das Hauptprogramm --
ANFANG:
  mov  DX,ANLEITUNG
  mov  AH,09
  int  021             ; Anleitungsausgabe
S1:
  mov  AH,010          ; Tastaturabfrage
  int  016
  mov  BUCHSTABE,AL    ; Zeichen in Ausgabetext einbauen
  push AX              ; Zeichen sichern
    mov  DI,ZEICHEN    ; Zeiger auf Stringanfang setzen
    call DEZCODE       ; Dezimalwert als ASCII-String wandeln
    mov  SI,TEXT       ; Stringadresse
    mov  CL,1          ; Spalte
    mov  CH,25         ; Zeile
    mov  AH,0E         ; Farbe
    call AUSGABE       ; Stringausgabe an Zeile/Spalte
  pop  AX              ; gesichertes Zeichen zurückholen
  cmp  AL,01B          ; wurde <Esc> gedrückt?
  jne  S1              ; wenn nein, weiter in Schleife
;-- Programmende:
  mov  AH,04C
  int  021
```

26.6 Aufgabe 4.4 (TPUFFER)

Zum Hauptprogramm ist rechts ein Flussdiagramm dargestellt. Zunächst wird der Bildschirm gelöscht, dann der Anleitungstext mit den Überschriften ausgegeben und schließlich wird das Segmentregister **DS** auf das BIOS-Datensegment an der Adresse **40hex** eingestellt. Dadurch können wir anschließend direkt auf die BIOS-Daten zugreifen. Danach beginnt die Hauptschleife.



Die Tastatur wird abgefragt. Wenn die Taste <**Esc**> gedrückt wurde, beendet sich das Programm. Anderenfalls wird eine Zeile mit Daten dargestellt. Dazu wird zunächst ein Zeilenumbruch gemacht. Dann wird der Lesezeiger **SI** auf den Anfang der Daten gestellt, das Zählregister **CX** auf 18 Schleifendurchläufe und das Richtungsflag für String-Operationen auf *vorwärts* eingestellt.

Die nachfolgende Zählschleife ab **S2:** wird nun 18 mal durchlaufen. Jedes Mal wird ein `word` aus dem Tastaturpuffer gelesen und als hexadezimaler Wert auf dem Bildschirm dargestellt. Die ersten beiden Werte sind der Lesezeiger und der Schreibzeiger des Tastaturpuffers. Wenn der Zähler auf 17 steht (es sind genau die beiden Zeigerwerte auf den Bildschirm geschrieben worden), dann wird zusätzlich ein Leerzeichen ausgegeben, damit die Zeigerwerte sich besser von den Daten abheben. Anschließend werden die 16 Zeichencodes dargestellt, die im Tastaturpufferbereich liegen.

Die Ausgabe der Werte in hexadezimaler Form erfolgt in einem Unterprogramm. Damit die Ziffern in der richtigen Reihenfolge auf den Bildschirm gebracht werden, muss zuerst das höherwertige Byte und darin das höherwertige Nibble ausgegeben werden. Zur Berechnung dieser Ziffern wird das Unterprogramm `HEXCODE` verwendet, das wir bereits früher im Programm **HTASTEN**aus Aufgabe 3.3a erstellt haben.

Es folgt das Listing des Programms.

```
; Programm TPUFFER
; Das Programm gibt eine Anleitung aus und stellt bei Tastendruck
; den aktuellen Inhalt des Tastaturpuffers an der Adresse
; 0040:001E bis 0040:003D dar.
; Mit der Taste <Esc> beendet sich das Programm.
jmp  START
ANLEITUNG:
  db 'Bitte eine Taste drücken!',0D,0A
  db 'Bei jedem Tastendruck wird der aktuelle Inhalt des '
  db 'Tastaturpuffers dargestellt.',0D,0A
  db 'Mit der Taste <Esc> wird das Program beendet.',0D,0A
  db ' Zeiger                            Daten',0D,0A
  db '!------! '
  db '!---------------------------------------------------------!$'
;-------------------------
;-- Prozedur HEXCODE wandelt Zeichen in AL in ASCII-Hexcode in AX
;   Dabei steht das höherwertige Zeichen in AH, das niederwertige in AL
HEXCODE:
  mov  AH,AL  ; Wert auch nach AH kopieren
  and  AL,0F  ; höherwertigen Teil in AL löschen
  shr  AH,4   ; höherwertigen Teil in AH isolieren
  add  AX,03030 ; beide Teile in ASCII-Zeichen wandeln
  cmp  AL,'9' ; ist Zahl bis einschließlich Ziffer 9?
  jbe  >L1    ; wenn ja, fertig, überspringen
  add  AL,7   ; sonst noch Korrekturwert addieren
L1:
  cmp  AH,'9' ; ist höherwertiger Teil Zahl bis einschließlich Ziffer 9?
  jbe  ret    ; wenn ja, fertig
  add  AH,7   ; sonst noch Korrekturwert addieren
ret
;------------------
AUSGABE:     ; Ausgabe des Zeichens in AL mit Funktion 0Eh des Int 10h
  mov  AH,0E ;
  int  010
ret
;------------------
TEXTAUSGABE:       ; Adresse des String in DX übergeben!
  mov  AH,09       ; Funktionsnummer für Textausgabe
  int  021
ret
;-------------------------
```

```
UMBRUCH:              ; Zeichen für Zeilenumbruch schreiben
  mov  AX,0E0D        ; Funktion "Zeichen schreiben", Zeichen 0D
  int  010            ; ausführen
  mov  AL,0A          ; Zeichen 0A
  int  010            ; ausführen
ret
;------------------
BILDSCHIRM_LOESCHEN:
  mov  AX,0B800
  mov  ES,AX          ; Bildschirmadresse einstellen
  mov  AX,0720        ; Leerzeichen als Löschzeichen, Farbe 7
  mov  CX,80*25       ; je 80 Zeichen in 25 Zeilen löschen
  xor  DI,DI          ; Zeiger auf 0 einstellen
  cld                 ; Richtung "vorwärts"
  rep  stosw          ; alles löschen
;-- nun Cursor zurücksetzen:
  xor  DX,DX          ; Cursor-zeile/ -spalte :=0
  mov  BL,DL          ; Bildschirmseite 0
  mov  AH,2           ; BIOS-Funktion Cursor setzen
  int  010            ; ausführen
ret
;------------------------
ZAHLENAUSGABE:        ; Die in AX übergebene Zahl wird als 4 Hex-Ziffern
                      ;   ausgegeben
  push AX             ; übergebene Zahl sichern
    mov  AL,AH        ; mit höherwertigem Byte beginnen
    call HEXCODE      ; oberes Byte in zwei Hex-Ziffern wandeln
    push AX           ; berechnete Ziffern sichern
      mov  AL,AH
      call AUSGABE    ; höchstwertige Ziffer ausgeben
    pop  AX           ; berechnete Ziffern zurückholen
    call AUSGABE      ; niederwertigen Teil (2. Hex-Ziffer) ausgeben
  pop  AX             ; übergebene Zahl zurückholen
  call HEXCODE        ; unteres Byte in zwei Hex-Ziffern wandeln
  push AX             ; berechnete Ziffern sichern
    mov  AL,AH        ;
    call AUSGABE      ; höherwertige Hex-Ziffer ausgeben
  pop  AX             ; berechnete Ziffern zurückholen
  call AUSGABE        ; niederwertige Hex-Ziffer ausgeben
ret
;------------------------
```

```
START:               ; Anfang Hauptprogramm
  call BILDSCHIRM_LOESCHEN
  mov  DX,ANLEITUNG
  call TEXTAUSGABE ; Anleitung ausgeben
  mov  AX,040
  mov  DS,AX       ; Datensegment auf BIOS-Datensegment einstellen!
  cld              ; Richtung "vorwärts"
S1:
  mov  AH,010
  int  016         ; Tastaturabfrage
  cmp  AL,01B      ; Esc gedrückt?
  jne  >L1         ; wenn nein, überspringen
  mov  AH,04C      ; sonst Programmende einleiten
  int  021
;---------------
L1:
  call UMBRUCH     ; Einen Zeilenumbruch machen
  mov  SI,01A      ; Zeiger auf Pufferanfang (Lese-Zeiger)
  mov  CX,18       ; Der Puffer umfasst 2 words für 2 Zeiger + 16 Zeichen
  cld              ; Richtung "vorwärts"
S2:
  lodsw            ; Zeichen aus Puffer holen
  call ZAHLENAUSGABE ; Ausgabe des Zeichens im Hex-Code
  cmp  CX,17       ; Ausgabe der beiden Zeiger soeben beendet?
  jne  >L2         ; wenn nein, überspringen
  mov  AL,' '
  call AUSGABE     ; nach 2 Zeichen ein Leerzeichen einfügen
L2:
  loop S2          ; Schleife für 32 Zeichen
  jmp  S1          ; und noch mal zur Tastaturabfrage
```

27 Musterlösungen, Teil 5

27.1 Aufgabe 5.1 (TON)

Das Programm war schon fast fertig. Für die neu einzutragenden Funktionen ist ein Flussdiagramm entbehrlich, Verzweigungen finden nicht statt. Daher folgt hier das Listing des fertigen Programms ohne weiteren Kommentar.

```
; Programm TON
; Das Programm erzeugt einen Ton, dessen Frequenz und Länge im
; Parameterstring übergeben wird. Beispiel: TON 800 500
; Im Beispiel würde ein Ton von 800Hz mit einer Länge von 500ms erzeugt.
  jmp  START
;-- Diverse Texte --------------------------------
ANLEITUNGSTEXT:
db 'Bitte dem Programm zwei Zahlenwerte für Frequenz (in Hertz) und',0A,0D
db 'Länge (in Millisekunden) übergeben. Beispiel für 800Hz und 500ms:',0A,0D
db 'TON 800 500',0A,0D,'$'
;--------------
FREQUENZFEHLER:
db 'Die als 1. Parameter angegebene Frequenz ist zu hoch!',0D,0A,'$'
;--------------
ZEITZFEHLER:
db 'Die als 2. Parameter angegebene Zeit ist zu groß!',0D,0A,'$'
;-- Variablen: -----------------------------------
FREQUENZ dw 0
LAENGE   dw 0
;==================================================
```

```
TONAUSGABE: ; schaltet Ton ein, dessen Frequenz in Hz in AX übergeben wird
;-- Zuerst wird der Teiler aus der Frequenz berechnet.
;   Dazu muß die Taktfrequenz von 1193180 (= 1234DC hex) durch die
;   übergebene Frequenz geteilt werden. Das Ergebnis ist der Teiler,
;   der dem Timerbaustein 8253 übergeben werden muß.
  mov  BX,AX         ; übergebene Frequenz nach BX übertragen
  mov  DX,012
  mov  AX,034DC      ; DX:AX := 1234DC (interne Taktfrequenz des 8253)
  div  BX            ; Teilen! Damit steht der Teiler in AX.
;-- Es folgt die Tonausgabe. Dazu zuerst Ausgaberegister für Timerbaustein
;   8253 vorbereiten:
  push AX            ; Teiler vorübergehend sichern
    mov  AL,0B6      ; Ausgaberegister für
    out  043,AL      ;   Timerbaustein 8253 vorber.
  pop  AX            ; Teiler zurückholen
;-- das niederwertige Byte übermitteln:
  out  042,AL        ; niederwertiges Byte übermitteln
;-- das höherwertige Byte übermitteln:
  mov  AL,AH         ;   (AX enthält Teilfaktor)
  out  042,AL        ; höherwertiges Byte übermitteln
;-- alten Status des 8253 holen:
  in   AL,061        ; alten Status 8253 holen
;-- und den Lautsprecher einschalten:
  or   AL,11xB       ; Lautsprecher
  out  061,AL        ;    einschalten
ret
;------------------
TON_AUS:  ; Den Ton wieder abschalten
;-- alten Status des 8253 holen:
  in   AL,061        ; alten Status 8253 holen
;-- und den Lautsprecher ausschalten:
  and  AL,11111101xB ; Lautsprecher
  out  061,AL        ;    ausschalten
ret
;------------------
```

```
WARTEN:    ; Wartet Zeit ab, die in AX (in ms) übergeben wird
  mov  CX,183         ; Anzahl der Schleifendurchläufe
  mul  CX             ;  = Zeit (in ms) * 183/10000
  mov  CX,10000
  div  CX
  mov  SI,AX          ; Zahl der notwendigen Schleifendurchläufe nach SI
  inc  SI             ; nach oben aufrunden
  mov  AX,0
  int  01A            ; Zahl der Timerticks holen
  mov  DI,DX          ; Zählerstand in DI merken
  mov  CX,SI          ; Zähler CX für Zählschleife setzen
S1:                   ; Warteschleife  1 Timertick
  push CX
    mov  AX,0
    int  01A          ; Zahl der Timerticks holen
  pop  CX
  cmp  DX,DI          ; wurde inzwischen hochgezählt?
  je   S1             ; wenn nein, kleine Warteschleife
  mov  DI,DX          ; neuen Wert in DI merken
  loop S1             ; Zählschleife
ret
;------------------
WARNTON:            ; Ausgabe eines Trillers als Warnton:
  mov  AX,670       ; Frequenz 670 Hz
  call TONAUSGABE
  mov  AX,100       ; Zeit 100 ms
  call WARTEN
  mov  AX,800       ; Frequenz 800 Hz
  call TONAUSGABE
  mov  AX,100       ; Zeit 100 ms
  call WARTEN
  mov  AX,670       ; Frequenz 670 Hz
  call TONAUSGABE
  mov  AX,100       ; Zeit 100 ms
  call WARTEN
  call TON_AUS      ; und Ton wieder abschalten
ret
;------------------
```

```
PRUEFE_ZIFFER: ; prüft, ob Zeichen in AL eine Ziffer ist.
              ; wenn ja, wird das ZF gesetzt, anderenfalls gelöscht.
  cmp  AL,'0'
  jb   ret    ; wenn AL kleiner '0', dann fertig mit gelöschtem ZF.
  cmp  AL,'9'
  ja   ret    ; wenn AL größer '9', dann fertig mit gelöschtem ZF.
  test AL,0C0 ; Zero-Flag setzen (die höchsten Bits von AL sind glöscht!)
ret
;------------------
PRUEFE_PARAMETER: ; prüft, ob 2 Parameter mit Ziffern übergeben wurden
                  ; wenn nein, wird das CF gesetzt, sonst gelöscht.
  mov  SI,080     ; Zeiger auf Parameterstring-Anfang einstellen
  cld             ; Richtungsflag für "vorwärts"
;-- Die Stringlänge wird nach CX geholt:
  lodsb           ; erstes Zeichen nach AL holen
  mov  CL,AL      ; Stringlänge nach CX laden
  mov  CH,0       ; (Dazu muß CH=0 sein)
  mov  DL,0       ; DL wird als Zähler für die Strings verwendet - Startwert
  jcxz >F0        ; wenn CX=0, dann fertig
;--
S1:
  lodsb              ; nächstes Zeichen holen
  call PRUEFE_ZIFFER ; prüft, ob Zeichen in AL eine Ziffer ist.
  loopne S1          ; CX runterzählen - Wenn keine Ziffer, weiter in Schleife
;-- Anfang von Ziffernstring gefunden --
  inc  DL            ; Zählen
  jcxz   >F0         ; wenn Stringende erreicht, zum Ende springen
S2:
  lodsb
  call PRUEFE_ZIFFER ; prüft, ob Zeichen in AL eine Ziffer ist.
  loope S2
  jcxz  >F0
  jmp   S1
;---------
F0:                  ; Ende
  cmp  DL,2          ; mindestens 2 Zahlen gefunden?
  jae  >F1           ; wenn ja, Sprung (fertig ohne Fehler)
  stc                ; "Fehler" markieren
ret
;--
F1:
  clc                ; "Kein Fehler" markieren
ret
;--------------------
```

```
STRING_IN_ZAHL: ; wandelt String bei DS:SI in Zahl
                ; Die Zahl wird in AX zurückgegeben
                ; Ist die Zahl zu groß, wird CF gesetzt.
  mov  CX,10    ; Faktor zum multiplizieren
  xor  AX,AX    ; Startwert: AX=0
  mov  BH,0
  cld           ; Richtungsflag für "vorwärts"
S1:
  xchg AX,BX    ; vorübergehender Tausch AX mit BX (wegen nachf. Befehl LODSB)
  lodsb         ; ein Zeichen holen
  call PRUEFE_ZIFFER ; prüft, ob Zeichen in AL eine Ziffer ist.
  xchg AX,BX    ; AX mit BX zurücktauschen
  jne  >L2      ; wenn keine Ziffer, dann fertig
  mul  CX       ; bisherigen Zahlenwert mal 10 nehmen
  cmp  DX,0     ; gibt es einen höherwertigen Teil, der nicht in AX passte?
  je   >L1      ; wenn nein, überspringen
  stc           ; sonst Fehler markieren
ret             ; und fertig
;--
L1:
  sub  BL,'0'   ; ASCII-Zeichen in Zahl wandeln
  add  AX,BX    ; aktuelle Ziffer zum Zahlenwert dazu addieren
  jc   ret      ; wenn jetzt ein Überlauf passierte, Abbruch mit Fehler
  jmp  S1
L2:
  clc           ; CF löschen, da Funktion ohne Fehler ablief
ret
;-------------------------------------------------------------------
START:
  call PRUEFE_PARAMETER ; prüft, ob 2 Parameter mit Ziffern übergeben wurden
  jc   ANLEITUNG        ; Im Fehlerfall Anleitung ausgeben und fertig
;-- Anfang des ersten Zahlenstrings suchen:
  mov  SI,081        ; Zeiger auf Anfang des Parameter-String einstellen
  cld                ; Richtungsflag für "vorwärts"
S1:
  lodsb              ; nächstes Zeichen holen
  call PRUEFE_ZIFFER ; prüft, ob Zeichen in AL eine Ziffer ist.
  jne  S1            ; wenn nein, weitersuchen in Schleife
```

```
;-- Anfang des ersten Zahlenstring gefunden - Umwandeln!
  dec  SI             ; Zeiger zurück auf den Anfang des Zahlenstring
  call STRING_IN_ZAHL ; wandelt String bei DS:SI in Zahl (in AX)
  jc   FREQUENZ_ZU_GROSS ; bei Überlauf mit Fehlermeldung abbrechen
  mov  FREQUENZ,AX    ; Frequenzwert abspeichern
;-- Anfang des zweiten Zahlenstring suchen:
S1:
  lodsb               ; nächstes Zeichen holen
  call PRUEFE_ZIFFER ; prüft, ob Zeichen in AL eine Ziffer ist.
  jne  S1             ; wenn nein, weitersuchen in Schleife
;-- Anfang des zweiten Zahlenstring gefunden - Umwandeln!
  dec  SI             ; Zeiger zurück auf den Anfang des Zahlenstring
  call STRING_IN_ZAHL ; wandelt String bei DS:SI in Zahl (in AX)
  jc   ZEIT_ZU_GROSS
  mov  LAENGE,AX      ; Zeitdauer abspeichern
;-- Frequenz und Dauer sind bestimmt. Jetzt kann der Ton ausgegeben werden!
  mov  AX,FREQUENZ    ; Frequenz (in Hertz) für die Tonausgabe laden
  call TONAUSGABE
  mov  AX,LAENGE      ; Zeit (in Millisekunden) für Tondauer laden
  call WARTEN
  call TON_AUS        ; und Ton wieder abschalten
  jmp  ENDE           ; das wars schon!
;-- Es folgen die Textausgaben für die jeweiligen Fehler:
FREQUENZ_ZU_GROSS:
  mov  DX,FREQUENZFEHLER
  jmp  AUSGABE
;--
ZEIT_ZU_GROSS:
  mov  DX,ZEITZFEHLER
  jmp  AUSGABE
;--
ANLEITUNG:
  mov  DX,ANLEITUNGSTEXT
AUSGABE:
  push DX
    call WARNTON
  pop  DX
  mov  AH,09          ; Text ausgeben
  int  021
ENDE:
  mov  AH,04C
  int  021
```

27.2 Aufgabe 5.2 (SCANCODE)

Die interessanten Befehle haben keine Verzweigungen, ein Flussdiagramm ist daher entbehrlich. Hier folgt das Listing:

```
; Programm SCANCODE
; Das Programm stellt den Scancode aller Tasten sowie die damit erzeugten
; Zeichen auf dem Bildschirm dar.
jmp  START
;--------
TEXT:       db '    Zeichen: '
ZEICHEN     db 'x Code: '   ; Hier wird das Zeichen eingetragen
ZEICHEN_H   dw 0       ; hier wird der Zeichencode (höherwertig) eingetragen
ZEICHEN_L   dw 0       ; hier wird der Zeichencode (niederwertig) eingetragen
            db 'hex$'
;--------
AUSGABE:     ; Ausgabe des Zeichens in AL mit Funktion 0Eh des Int 10h
  mov  AH,0E ; Das Unterprogramm wird nur von der Interruptroutine verwendet.
  int  010
ret
;-- Prozedur HEXCODE wandelt Zeichen in AL in ASCII-Hexcode in AX
;   Dabei steht das höherwertige Zeichen in AH, das niederwertige in AL
HEXCODE:
  mov  AH,AL  ; Wert auch nach AH kopieren
  and  AL,0F  ; höherwertigen Teil in AL löschen
  shr  AH,4   ; höherwertigen Teil in AH isolieren
  add  AX,03030 ; beide Teile in ASCII-Zeichen wandeln
  cmp  AL,'9' ; ist Zahl bis einschließlich Ziffer 9?
  jbe  >L1    ; wenn ja, fertig, überspringen
  add  AL,7   ; sonst noch Korrekturwert addieren
L1:
  cmp  AH,'9' ; ist Zahl bis einschließlich Ziffer 9?
  jbe  ret    ; wenn ja, fertig
  add  AH,7   ; sonst noch Korrekturwert addieren
ret
;-------
NEUER_INT09:  ; Zusätzliche vorgeschaltete Interruptroutine für Int. 09h
;-- Register sichern, die in der Routine benötigt werden:
  push AX
;-- die Zeichen 0Dh und 0Ah für Zeilenende/neue Zeile ausgeben:
    mov  AL,0D      ; Ausgabe 0D
    call AUSGABE
    mov  AL,0A      ; Ausgabe 0A
    call AUSGABE
;-- Scancode von der Tastatur (aus Port 60h) nach AL holen:
```

```
    in   AL,060
;-- Scancode in Hexcode umwandeln:
;   (Hierzu darf das gleiche Unterprogramm wie das vom Hauptprogramm
;    benutzte verwendet werden.)
    call HEXCODE
;-- Höherwertiges Byte des ASCII-Zeichens ausgeben:
    push AX
      mov  AL,AH
      call AUSGABE
    pop  AX
;-- Niederwertiges Byte des ASCII-Zeichens ausgeben:
    call AUSGABE
;-- gesicherte Werte in Register zurückspeichern:
  pop  AX
;-- Zur ursprünglichen Interrupt-Routine springen:
  jmp  0:0   ; Die korrekte Adresse wird zur Laufzeit "eingepatcht".
ALTER_INT09 EQU $-4 ; (wird zum Einpatchen der Adresse benötigt)
;-- Hauptprogramm: ----------------------------------------------
START:
;-- Adresse des alten Interrupt 09h ermitteln:
  mov  AX,03509
  int  021
;-- alte Adresse in Sprungadresse am Ende der Ergänzungsroutine einpatchen:
  mov  W[ALTER_INT09],BX
  mov  W[ALTER_INT09+2],ES
;-- Adresse des neuen Interrupt 09h in Interrupttabelle eintragen:
  mov  DX,NEUER_INT09
  mov  AH,025
  int  021
S1:                 ; Hauptschleifenanfang
;-- Zeichen von der Tastatur holen:
  mov  AH,010
  int  016
;--
  mov  ZEICHEN,AL   ; Zeichen in Text eintragen
  push AX           ; Zeichencode sichern
    mov  AL,AH      ; höherwertiges Byte nach AL zur Umwandlung
    call HEXCODE    ;   und in ASCII Hexcode umwandeln
    xchg AL,AH      ; Bytes ordnen
    mov  ZEICHEN_H,AX ; und in Text eintragen
  pop  AX           ; Zeichencode zurückholen
  call HEXCODE      ; niederwertiges Byte in ASCII-Hexcode umwandeln
  xchg AL,AH        ; Bytes ordnen
  mov  ZEICHEN_L,AX ;   und in Text eintragen
```

```
;-- Ausgabe des Textes:
  mov  DX,TEXT
  mov  AH,09
  int  021
;-- Wiederholen, wenn nicht <Esc> gedrückt wurde:
  cmp  ZEICHEN,01B
  jne  S1           ; Schleife, bis <Esc> gedrückt wird
ENDE:
;-- Interrupt 09 wieder auf ursprünglichen Wert zurückstellen:
  lds  DX,ALTER_INT09
  mov  AX,02509
  int  021
;-- Programmende:
  mov  AH,04C
  int  021
```

28 Musterlösungen, Teil 6:

28.1 Aufgabe 6.1 (ASTEST)

Hier folgt das Listing des fertigen Testprogramms `ASTEST.PAS`.

```
PROGRAM ASTEST;  {Demonstration zum Einbinden von Assembler-Routinen}

Var Cursor_X, Cursor_Y: Byte;
    Zeichen: Char;
    i: Byte;

{Es folgen die externen Funktionen und Prozeduren:}
PROCEDURE Cursor_Setzen; EXTERNAL;
  {Setzt Cursor an Stelle CURSOR_X/CURSOR_Y}
PROCEDURE Bildschirm_Loeschen(Loeschzeichen: Char; Farbe: Byte); EXTERNAL;
  {Überschreibt Bildschirm mit LOESCHZEICHEN in der Farbe FARBE}
FUNCTION Hole_Taste:Char; EXTERNAL;  {Holt ein Zeichen von der Tastatur}
PROCEDURE Schreib(X, Y, Farbe: BYTE; Str: STRING); EXTERNAL;
{$L ASTEST.OBJ}  {Anweisung an Compiler: Hier sind die Prozeduren zu finden.}

Begin
  Bildschirm_Loeschen(' ',$1E); {Löschen mit blauem Hintergrund}
  Schreib(22,5,$4F,'*-----------------------------------*');
  for i:=6 to 15 do Schreib(22,i,$4F,'!                                   !');
  Schreib(22,16,$4F,'*-----------------------------------*');
  Schreib(35,8,$4E,'Schreibtest');
  Schreib(31,12,$4E,'Abbrechen mit <Esc>');
  REPEAT
    Zeichen:=Hole_Taste;
  UNTIL Zeichen = #27;
  Bildschirm_Loeschen(' ',$07); {Löschen in Standard-Farbe}
  Cursor_X := 1;
  Cursor_Y := 1;
  Cursor_Setzen;        {Cursor nach oben links}
END.
```

Hier folg das Listing von ASTEST.ASM.

```
; Hilfsroutinen für PASCAL-Programm ASTEST.PAS
; Compilieren mit dem Befehl:
;
;   A386 ASTEST.8 +O
; oder:
;   A386 ASTEST.8 ASTEST.OBJ
;
;==========================================
CODE SEGMENT BYTE PUBLIC      ; compilieren für PASCAL
EXTRN CURSOR_X:B, CURSOR_Y:B  ; Byte-Variablen, sind unter PASCAL definiert
;==========================================
CURSOR_SETZEN:     ; Cursor auf Position CURSOR_X/CURSOR_Y setzen
  mov  AH,2        ; Funktionsnummer
  mov  BH,0
  mov  DL,CURSOR_X ; unter PASCAL definierte Variable lesen
  mov  DH,CURSOR_Y
  dec  DH          ; Zählweise
  dec  DL          ;    anpassen
  int  010
ret
;==========================================
BILDSCHIRM_LOESCHEN:
; Überschreibt den Bildschirm (25 Zeilen zu 80 Zeichen)
; mit dem übergebenen Löschzeichen in übergebener Farbe.
  push BP              ; BP sichern
    mov  BP,SP         ; Basiszeiger "verankern"
    mov  AX,0B800      ; ES auf Bildschirmsegment
    mov  ES,AX         ;    einstellen
    mov  AL,[BP+6]     ; übergebenes Lösch-Zeichen nach AL einlesen
    mov  AH,[BP+4]     ; übergebene Farbe nach AH einlesen
    mov  CX,80*25      ; Zahl der Bildschirmzeichen
    cld                ; Richtung auf "Vorwärts"
    xor  DI,DI         ; beim ersten Bildschirmzeichen anfangen
    rep  stosw         ; alles überscheiben
  pop BP               ; BP restaurieren
ret 4
;==========================================
HOLE_TASTE:            ; Holt ein Zeichen vom Tastaturpuffer ab
  mov  AH,010
  int  016             ; das Zeichen wird in AL geliefert, das passt
ret
;==========================================
```

```
SCHREIB:
  push BP
    mov  BP,SP          ; Hilfszeiger zum Lesen auf dem Stack festlegen
    mov  AX,0B800
    mov  ES,AX          ; Bildschirmadresse einstellen
    mov  BH,[BP+8]      ; Farbe
    push DS
      lds  SI,[BP+4]    ; String-Adresse nach DS:SI laden
      mov  CH,0
      mov  CL,[SI]      ; Stringlänge nach CX
      jcxz ENDE         ; Rücksprung, wenn String leer
      mov  DH,0
      mov  DL,[BP+12]   ; Pos X
      mov  DI,DX
      dec  DI           ; Zählweise anpassen
      mov  AX,80
      mov  BL,[BP+10]   ; Pos Y
      dec  BL,1         ; Zählweise anpassen
      mul  BL           ; AX:=y*80
      add  DI,AX        ; DX:=y*80+x
      shl  DI,1         ; DI:= 2*DX (Wortlänge!)
      mov  AH,BH        ; Farbe
      cld
      inc  SI           ; (hinter Längenbyte anfangen)
AUSGABE:
      lodsb             ; Hole !Byte!
      stosw             ; Schreibe !Wort! incl. Farbe
      loop AUSGABE      ; Schreibschleife
ENDE:
    pop  DS
  pop  BP
ret 10                  ; Rücksprung, 5 Worte vom Stapel abräumen
```