Lehrstuhl für Automatisierung und Informationssysteme
Prof. Dr.-Ing. B. Vogel-Heuser

Vorname:	
Nachname:	
Matrikelnummer:	

Prüfung

Grundlagen der modernen Informationstechnik

Sommersemester 20[illegible]

28.08.2023

Bitte legen Sie Ihren Lichtbildausweis bereit.

Sie haben für die Bearbeitung der Klausur 120 Minuten Zeit.

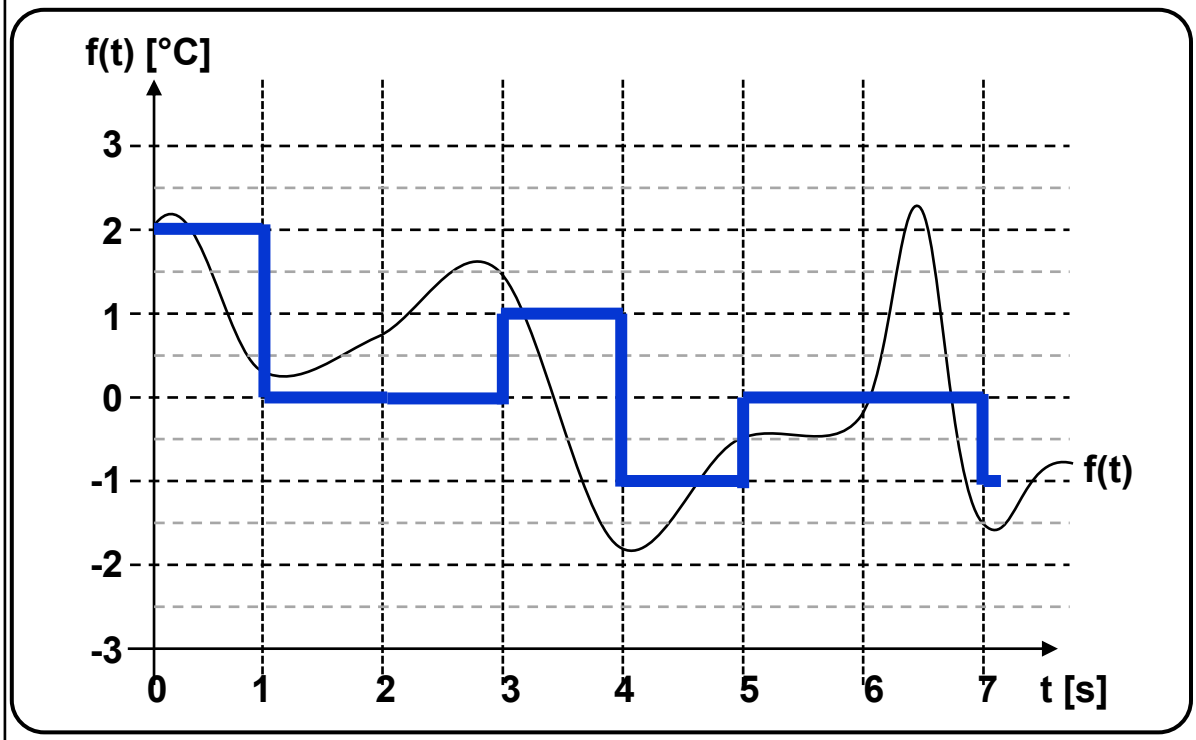
Diese Prüfung enthält 35 nummerierte Seiten inkl. Deckblatt.

Bitte prüfen Sie die Vollständigkeit Ihres Exemplars!

Bitte nicht mit rot oder grün schreibenden Stiften oder Bleistift ausfüllen!

Aufgabe 1: Grundlagen der Informationstechnik und Digitaltechnik

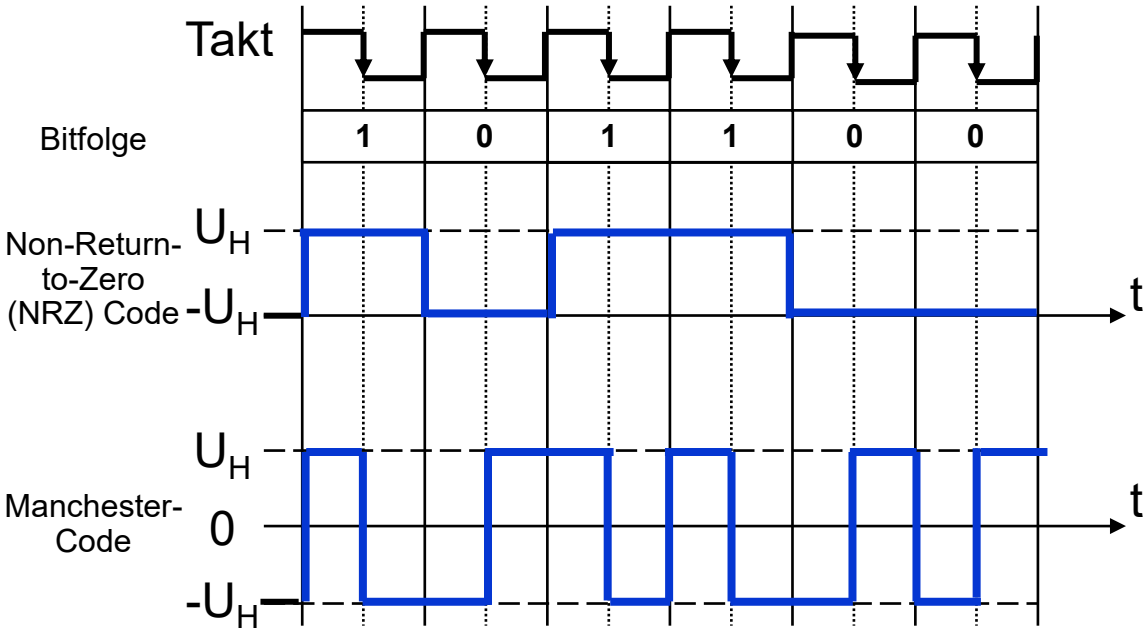
a) Gegeben ist das dargestellte, wert- und zeitkontinuierliche Signal f(t). Zeichnen Sie den wertdiskreten (aber zeitkontinuierlichen) Signalverlauf von f(t) für $t \in [0;7]$ in das Schaubild ein. Die Diskretisierung erfolgt jede Sekunde [s] auf ganzzahlige °C, wobei die Nachkommastellen der Werte vernachlässigt werden (d.h. 2.8 → 2).



Aufgabe 1: Grundlagen der Informationstechnik und Digitaltechnik

b) Sie versenden die **Bitfolge `101100`** auf einem seriellen Bussystem. Zeichnen Sie den resultierenden Leitungscode als Non-Return-To-Zero und im Manchester-Code. Für beide Codes liegt **zu Beginn $-U_H$** an.

Lösung für Bitfolge 101100



Aufgabe 2: IEEE 754 Gleitkommadarstellung und Zahlensysteme

a) Rechnen Sie die Dezimalzahl $\mathbf{(-12{,}25)_{10}}$ in eine Gleitkommazahl (angelehnt an die IEEE 754 Darstellung) um, indem Sie die folgenden Textblöcke ausfüllen.

Hinweis: Ergebnisse und Nebenrechnungen außerhalb der dafür vorgesehenen Textblöcke werden nicht bewertet!

Vorzeichen V 1 Bit	Biased Exponent E 3 Bit	Mantisse M 5 Bit

Vorzeichenbit

1

Dezimalzahl $(12{,}25)_{10}$ als Binärzahl

1100,01

Bias als Dezimalzahl

$$B = 2^{(x-1)} - 1 = 2^{(3-1)} - 1 = (3)_{10}$$

Exponent als Dezimalzahl

$$e = (3)_{10}$$

Biased Exponent als Dualzahl

$$E = e + B = 3 + 3 = (6)_{10} = (110)_2$$

Vollständige Gleitkommazahl (nach obigem Schema)

1	110	10001
Vorzeichen	**Biased Exponent**	**Mantisse**

b) Überführen Sie die unten gegebenen Zahlen in die jeweils anderen Zahlensysteme.
Hinweis: Achten Sie genau auf die jeweils angegebene Basis.

1 $\quad (111\,1101)_2 = (\underline{\quad 175 \quad})_8 = (\underline{\quad 7D \quad})_{16}$

2 $\quad (132)_5 = (\underline{\quad 42 \quad})_{10}$

Aufgabe 3: Logische Schaltungen und Schaltbilder

a) Vervollständigen Sie das Schaltbild, um die vollständige disjunktive Normalform (DNF) passend zur Wahrheitstabelle (Tabelle 3.1) zu erhalten.

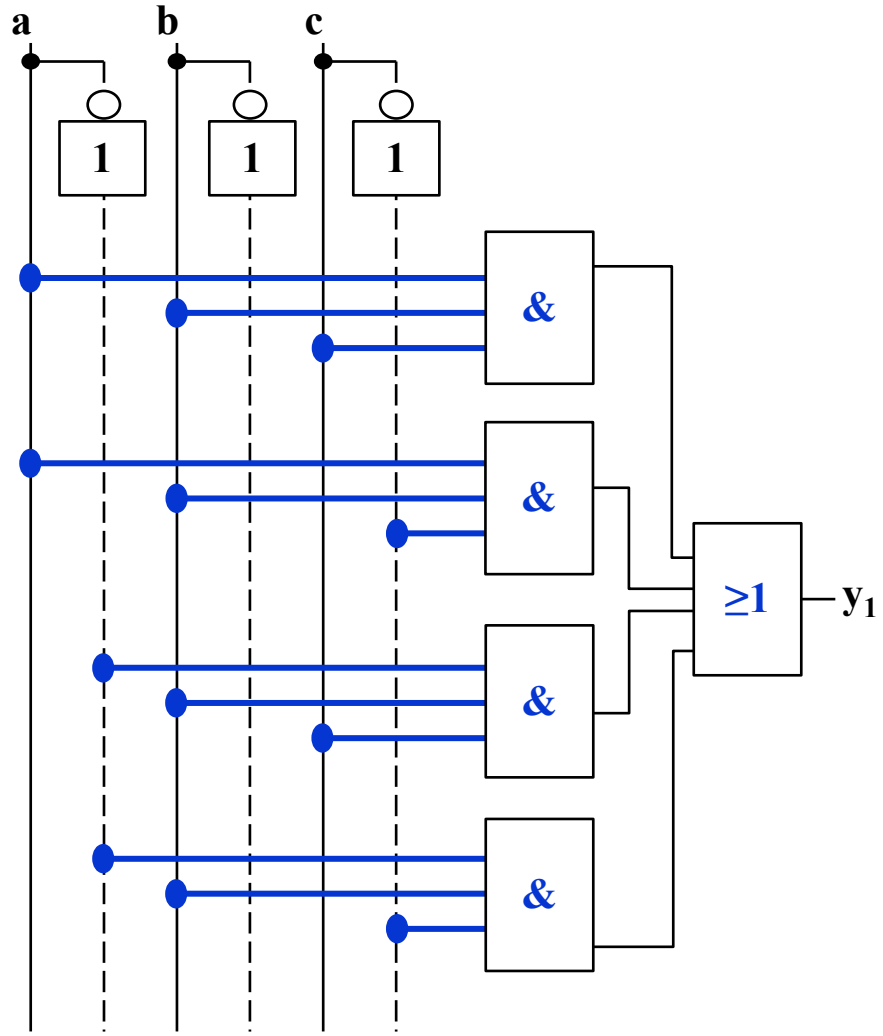


Tabelle 3.1:
Wahrheitstabelle

a	b	c	y_1
1	1	1	1
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	1
0	0	1	0
0	0	0	0

b) Stellen Sie für die Wahrheitstabelle (Tabelle 3.1) die zugehörige Disjunktive Normalform (DNF)-Gleichung auf und geben Sie die minimierte Schaltgleichung an.

Disjunktive Normalform (DNF):

$$y1 = (a \wedge b \wedge c) \vee (a \wedge b \wedge \bar{c}) \vee (\bar{a} \wedge b \wedge c) \vee (\bar{a} \wedge b \wedge \bar{c})$$

Alt.:

$$y1 = (a\ b\ c) \vee (a\ b\ \bar{c}) \vee (\bar{a}\ b\ c) \vee (\bar{a}\ b\ \bar{c})$$

Minimierte Schaltgleichung:

$$y1 = b$$

Aufgabe 4: FlipFlops

Gegeben ist die folgende Master-Slave-Flip-Flop-Schaltung (Bild 4.1):

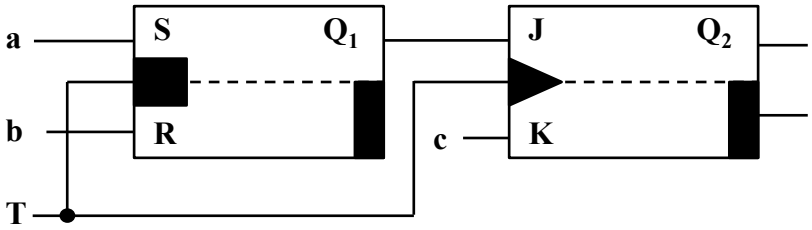
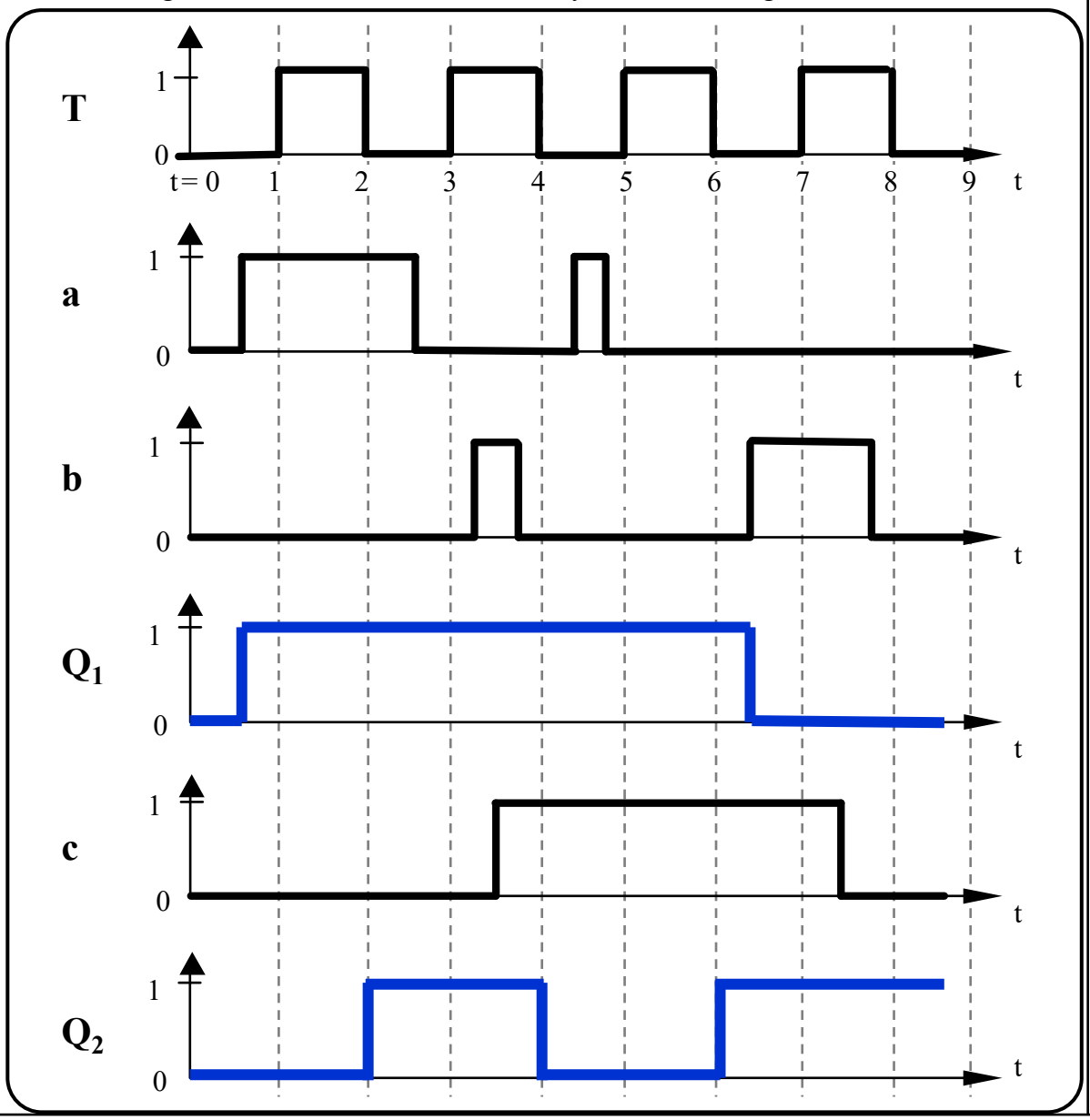


Bild 4.1: *MS-FF*

Bei t = 0 sind die Flip-Flops in folgendem Zustand: Q1 = Q2 = 0.

Analysieren Sie die Schaltung für den Bereich t=[0;9[, indem Sie für die Eingangssignale a, b, c und T die zeitlichen Verläufe für Q1 und Q2 in die vorgegebenen Koordinatensysteme eintragen.

Hinweis: Signallaufzeiten können bei der Analyse vernachlässigt werden.



Aufgabe 5: Automaten

a) Gegeben ist der in Bild 5.1 gezeigte Automat. Kreuzen Sie für I. bis III. die jeweils zutreffende Aussage an.

Der in Bild 5.1 gezeigte Automat …

I. ( ) … ist ein Mealy-Automat
(X) … ist ein Moore-Automat

II. ( ) … ist deterministisch
(X) … ist nicht-deterministisch

III. (X) … hat den Startzustand S1
( ) … hat den Startzustand S2
( ) … hat den Startzustand S3

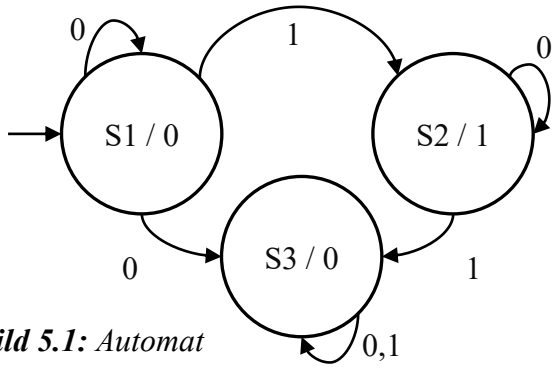


Bild 5.1: *Automat*

b) Erstellen Sie zu dem abgebildeten Automaten die zugehörige Übergangstabelle.

T	S1,0	S2,1	S3,0
0	**S1, S3**	**S2**	**S3**
1	**S2**	**S3**	**S3**

c) Welche Ausgabe erhalten Sie für die Eingabe 10011, wenn sich der Automat zu Beginn im Zustand S1 befindet?

1 1 1 0 0 **0 11100 as well correct**

d) Geben Sie die Zustandsübergänge A, B und C an, damit der Automat (Bild 5.2) der Übergangstabelle (Tabelle 5.1) entspricht.

Tabelle 5.1: *Übergangstabelle*

T	Z1	Z2
0	Z2 , 0	Z2 , 1
1	Z2 , 1	Z1 , 0

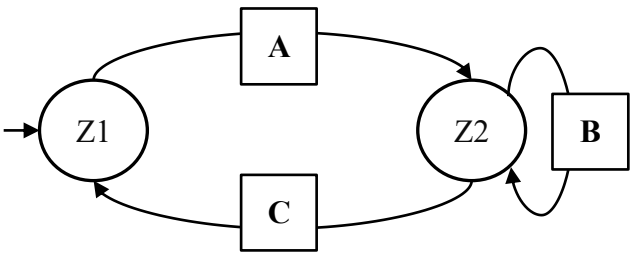


Bild 5.2: *Automat mit fehlenden Übergängen*

A	**0 / 0, 1 / 1**	**alt.: 1 / 1 , 0 / 0**
B	**0 / 1**	
C	**1 / 0**	

Aufgabe 6: MMIX – Assembler-Code

Gegeben sei der nachfolgende Algorithmus sowie die Ausschnitte der MMIX-Code-Tabelle (Bild 6.1) und eines Registerspeichers (Bild 6.2).

	0x_0	0x_1	…	0x_4	0x_5	…
	0x_8	0x_9		0x_C	0x_D	
…	…	…	…	…	…	…
0x1_	FMUL	FCMPE	…	FDIV	FSQRT	…
	MUL	MUL I		DIV	DIV I	
0x2_	ADD	ADD I	…	SUB	SUB I	…
	2ADDU	2ADDU I		8ADDU	8ADDU I	
…	…	…	…	…	…	…
0x8_	LDB	LDB I	…	LDW	LDW I	…
	LDT	LDT I		LDO	LDO I	
0x9_	LDSF	LDSF I	…	CSWAP	CSWAP I	…
	LDVTS	LDVTS I		PREGO	PREGO I	
0xA_	STB	STB I	…	STW	STW I	…
	STT	STT I		STO	STO I	
…	…	…	…	…	…	…
0xE_	SETH	SETMH	…	INCH	INCMH	…
	ORH	ORMH		ANDNH	ANDNMH	

Bild 6.1: *MMIX-Code-Tabelle*

Algorithmus:

$$x = \left(\frac{a}{(b-5) \bullet c} + 16\right) \bullet a$$

Registerspeicher		
Adresse	Wert vor Befehlsausführung	Kommentar
…	…	…
$0x86	0x00 00 00 00 00 00 62 0F	Zwischen-ergebnis
$0x87	0x00 00 00 00 00 00 AF FE	Variable a
$0x88	0x00 00 00 00 00 00 00 01	Variable b
$0x89	0x00 00 00 00 00 00 00 01	Variable c
…	…	

Bild 6.2: *Registerspeicher*

Im Registerspeicher eines MMIX-Rechners befinden sich zu Beginn die in Bild 6.2 gegebenen Werte. In der Spalte Kommentar wurde angegeben, welche Daten diese enthalten und wofür die einzelnen Zellen benutzt werden müssen.

a) Führen Sie den gegebenen Algorithmus aus. Verwenden Sie dazu lediglich die in Bild 6.1 **umrahmten Befehlsbereiche**. **Speichern Sie die Zwischenergebnisse** nach jedem Befehl des Algorithmus in der Registerzelle mit dem Kommentar **Zwischenergebnis**. Übersetzen Sie die Operationen in **Assembler-Code mit insgesamt maximal 5 Anweisungen**.

1	SUBI $0x86, $0x88 0x05	Alt: SUBI $0x86, $0x88 0x05
2	MUL $0x86, $0x86, $0x89	DIV $0x86, $0x87, $0x86
3	DIV $0x86, $0x87, $0x86	DIV $0x86, $0x86, $0x89
4	ADDI $0x86, $0x86, 0x10	ADDI $0x86, $0x86, 0x10
5	MUL $0x86, $0x86, $0x87	MUL $0x86, $0x86, $0x87

b) Sie möchten Variable a (Bild 6.3, links) des Registerspeichers in den Datenspeicher (Bild 6.3, rechts) als Byte an die Adresse M[0x00 … 00 05] speichern.

Hinweis: Für diese Teilaufgabe gilt die Big-Endian Adressierung.

Registerspeicher		
Adresse	Wert vor Befehlsausführung	Kommentar
$0x86	0x00 00 00 00 00 00 00 01	Wert 1
$0x87	0x00 00 00 00 00 00 00 02	Wert 2
$0x88	0x00 00 00 00 00 00 00 03	Wert 3
$0x89	0x00 00 00 00 00 00 00 0A	Variable a

Datenspeicher	
Adresse	Wert
M[0x00 … 00 03]	0x02
M[0x00 … 00 04]	0x04
M[0x00 … 00 05]	0x0C
M[0x00 … 00 06]	0x0A
M[0x00 … 00 07]	0x0F
M[0x00 … 00 08]	0x0E

Bild 6.3: *Register- und Datenspeicher*

Geben Sie den dafür benötigten Assembler-Befehl an. Nutzen Sie für die Adressierung **ausschließlich Werte des Registerspeichers** (Bild 6.3, links), keine Sofortoperanden:

STB $0x89 $0x87 $0x88

Sie führen den Befehl **LDT $0x89 $0x86 $0x87** aus. Ab welcher Adresse und bis zu welcher Adresse des Datenspeichers wird ausgelesen? Geben Sie die beiden Adressen, die gerade noch mit gelesen werden, an.

M[0x00 00 00 00 00 00 00 00]
M[0x00 00 00 00 00 00 00 03]

Wie viele Bits werden mit dem Befehl **LDT** geladen?

4*8 Bits = 32 Bits

c) Übersetzen Sie den folgenden Befehl **MUL (ohne Berücksichtigung von Parametern)** aus Assembler-Code in Maschinensprache (Hexadezimal) und rechnen Sie diese Hexadezimalzahl in eine Dezimal- und eine Binärzahl um.

Assemblersprache: **MUL**
Maschinensprache (Hexadezimal): 0x18
Dezimalzahl: 24
Binärzahl: 11000

d) Kreuzen Sie für die zwei folgenden Aussagen an, ob diese jeweils wahr oder falsch sind.

	Wahr	Falsch
1. Bei der direkten Adressierung ist der Wert des Operanden die gewünschte Speicheradresse.	(X)	( )
2. Der Datenspeicher dient der kurzfristigen Datenspeicherung.	( )	(X)

Aufgabe 7: Echtzeitprogrammiersprache PEARL

Vervollständigen Sie den untenstehenden Codeausschnitt in PEARL gemäß den Kommentaren über den Lücken. Der Codeausschnitt entstammt einer Etikettierstation, welche Flaschen etikettiert.

```
// Definieren Sie die Semaphore "Etikettierstation" mit
Startwert 1.
____DCL____Etikettierstation__SEMA PRESET 1_____ ;
// Definieren Sie die Unterbrechung "Kleber_leer".
___SPC Kleber_leer INTERRUPT_____ ;
// Definieren Sie den Task "Kleber_nachfuellen" mit Priorität 1.
 Kleber_nachfuellen:__TASK PRIORITY 1_____ ;
          Fuellen_starten;
END;
// Fragen Sie die Semaphore "Etikettierstation" an.
____REQUEST_____Etikettierstation;
// Ein Sensor gibt an, dass der Kleber leer ist. Aktivieren Sie
die Unterbrechung "Kleber_leer".
__ENABLE Kleber_leer_____;
// Wenn die Unterbrechung "Kleber_leer" eintritt, soll der Task
"Kleber_nachfuellen" aktiviert werden.
__WHEN Kleber_leer_____
      ___ACTIVATE___ Kleber_nachfuellen;
// Geben Sie die Semaphore "Etikettierstation" frei.
_RELEASE_____Etikettierstation;
END;
```

Aufgabe 8: Scheduling und Semaphoren

Gegeben sei der Soll-Verlauf der vier Prozesse A-D (Bild 8.1), welche **nach dem Earliest-Deadline-First-Prinzip präemptiv** für den Zeitraum $t = [0; 9]\,s$ und einem Zeitschlitz von einer Sekunde auf einem Einkernprozessor (CPU) ausgeführt werden sollen.

Beachten Sie, dass neue Tasks **immer nur zur vollen Sekunde** beginnen können. **Beginn** für das Scheduling ist der Zeitpunkt **t=0**.

Geben Sie für den Zeitraum t = [0;9] s die Reihenfolge der Tasks auf der CPU (Ist-Verlauf der Tasks) an.

Geben Sie zudem an, **wie viele Deadlines gehalten** werden.

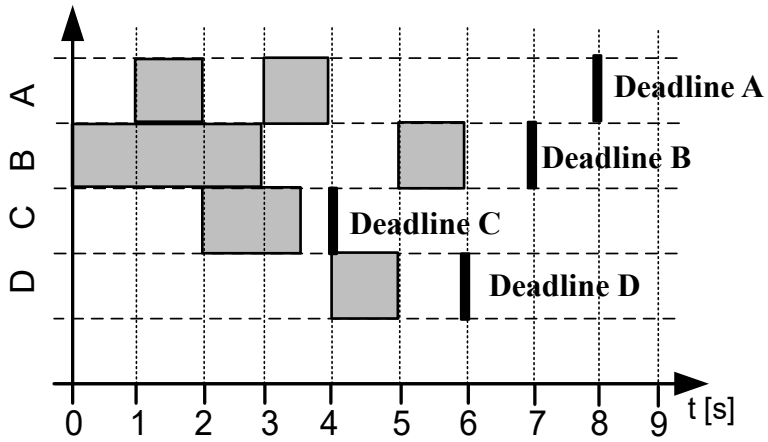
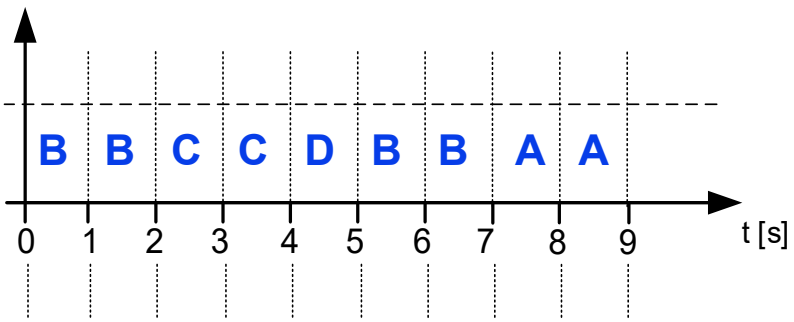


Bild 8.1: *Einplanung / Soll-Verlauf der Tasks*

Reihenfolge der Tasks auf der CPU (Ist-Verlauf der Tasks):

präemptiv



Es werden 3 **Deadlines gehalten.**

Aufgabe 9: IEC 61131-3 Funktionsbausteinsprache

Nachfolgend wird die Bedienung einer Intralogistik-Anlage beschrieben.

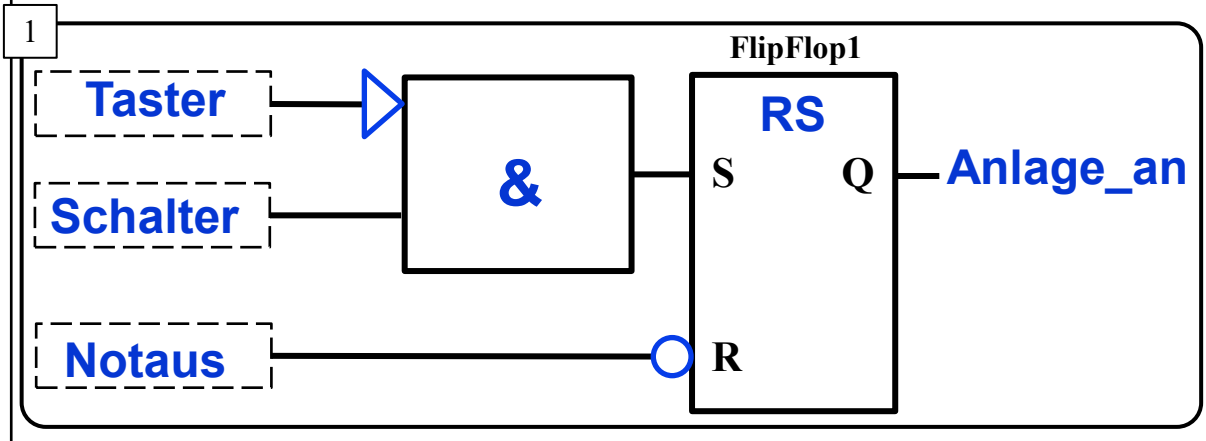
1
- Das Bedienpanel verfügt über einen Aktivierungstaster (**`Taster`**), einen Stromschalter (**`Schalter`**) sowie einen Notaus-Schalter (**`Notaus`**).
- Um die Anlage zu starten (**`Anlage_an`**), müssen sowohl der Aktivierungstaster (**`Taster`**) als auch der Stromschalter (**`Schalter`**) gedrückt bzw. geschaltet werden.
- Ein Notaus-Schalter (**`Notaus`**) stoppt die Anlage. Damit die Anlage trotz Stromausfall oder Kabelbruch stoppt, ist der Wert von **`Notaus`** dauerhaft 1 und wird 0 beim Schalten (oder im Falle eines Stromausfalls).
- **`Notaus`** hat immer Vorrang.
- **`Taster`** hat den Wert 1, wenn er gedrückt wird, und soll bei Betätigung die Anlage starten. Verhindern Sie, dass ein dauerhaftes gedrückt halten des Knopfes die Anlage wiederholt startet.

2
- Der Notaus-Schalter (**`Notaus`**) wird durch Betätigung eines Quittier-Schalters (**`Quittier`**) wieder auf 1 gesetzt. **`Quittier`** hat bei der Betätigung den Wert 1.

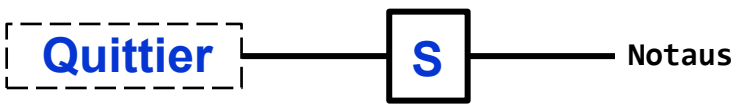
Ergänzen Sie die untenstehenden Programme [1] und [2] so, dass das oben beschriebene Verhalten erfüllt wird.

Hinweise:
- Signalverzögerungen im System sind zu vernachlässigen.
- Verwenden Sie **keine** Schaltglieder außer den in der Vorlage bereits vorhandenen.
- **Ergänzen Sie Negationen und Flankenerkennung falls notwendig**.

1



2



Aufgabe 10: UML-Use-Case-Diagramm

Ein Süßigkeitenhersteller ermöglicht durch eine **`Abfüllanlage`** die Zusammenstellung individueller Gummibärchentüten (Bild 10.1).

Gemeinsam mit dem Hersteller identifizieren Sie folgende Use Cases:

1. **`Kunden`** konfigurieren ihr Rezept an der **`Abfüllanlage`**.
2. Die Konfiguration des Rezepts enthält sowohl das Auswählen der Farbe der Gummibärchen (**`Farbe wählen`**) als auch das Auswählen des zugehörigen Gewichts (**`Gewicht wählen`**).
3. Optional kann bei der Konfiguration des Rezepts angegeben werden, ob an der Tüte ein personalisierter Aufdruck angebracht werden soll (**`Aufdruck wählen`**).

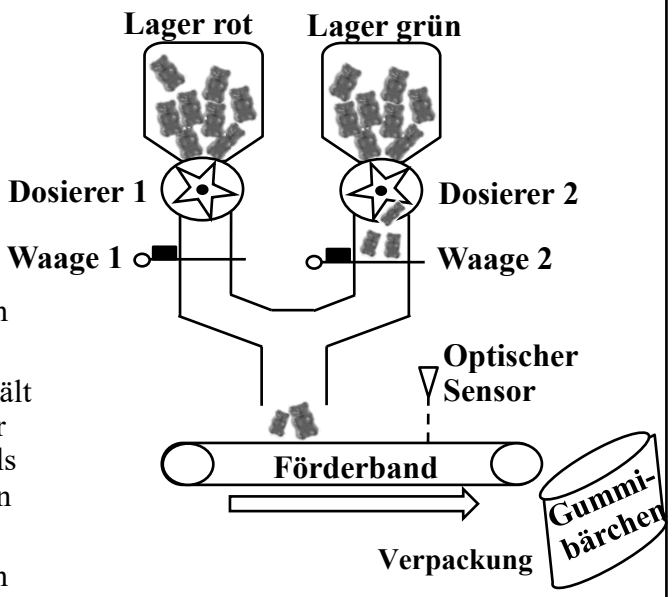
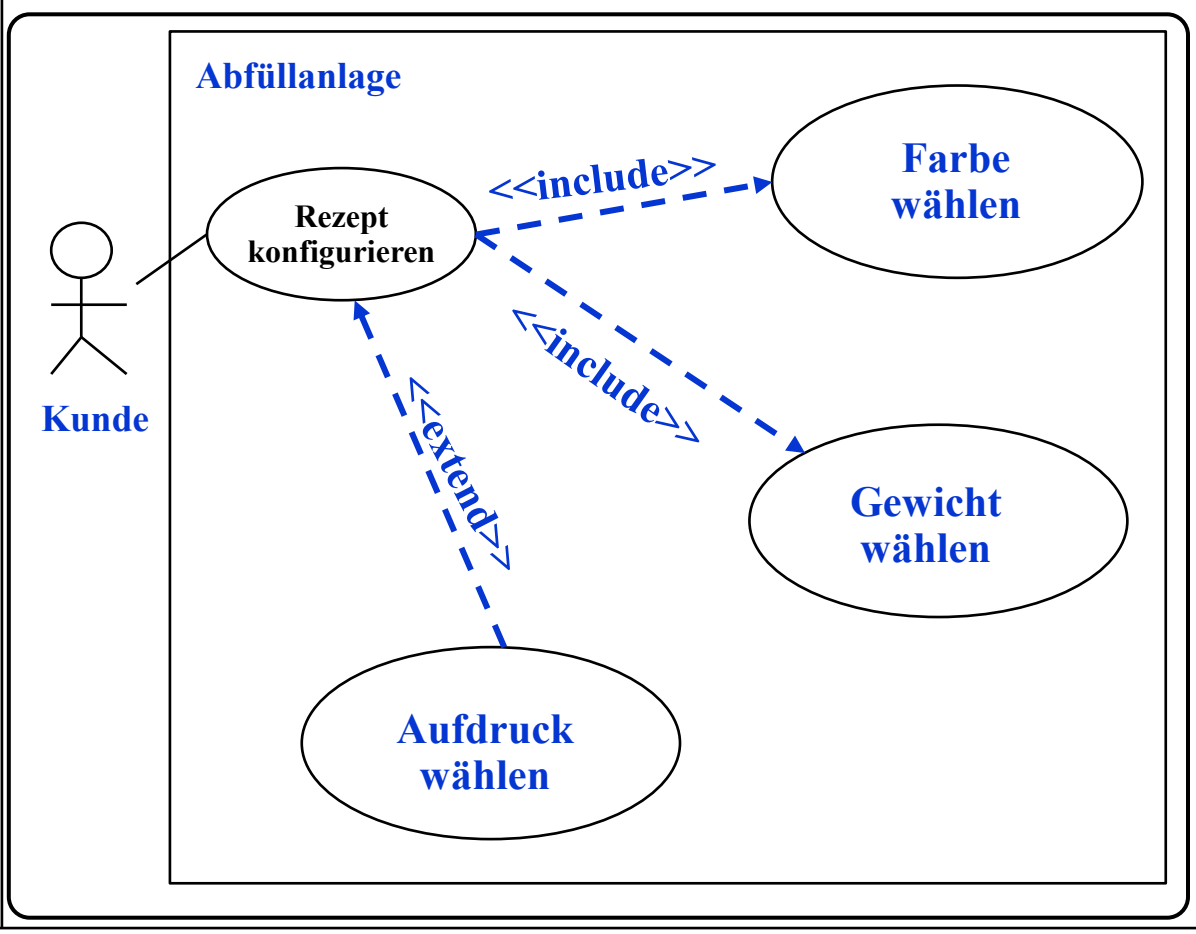


Bild 10.1: *Schematischer Aufbau der Abfüllanlage für Gummibärchen*

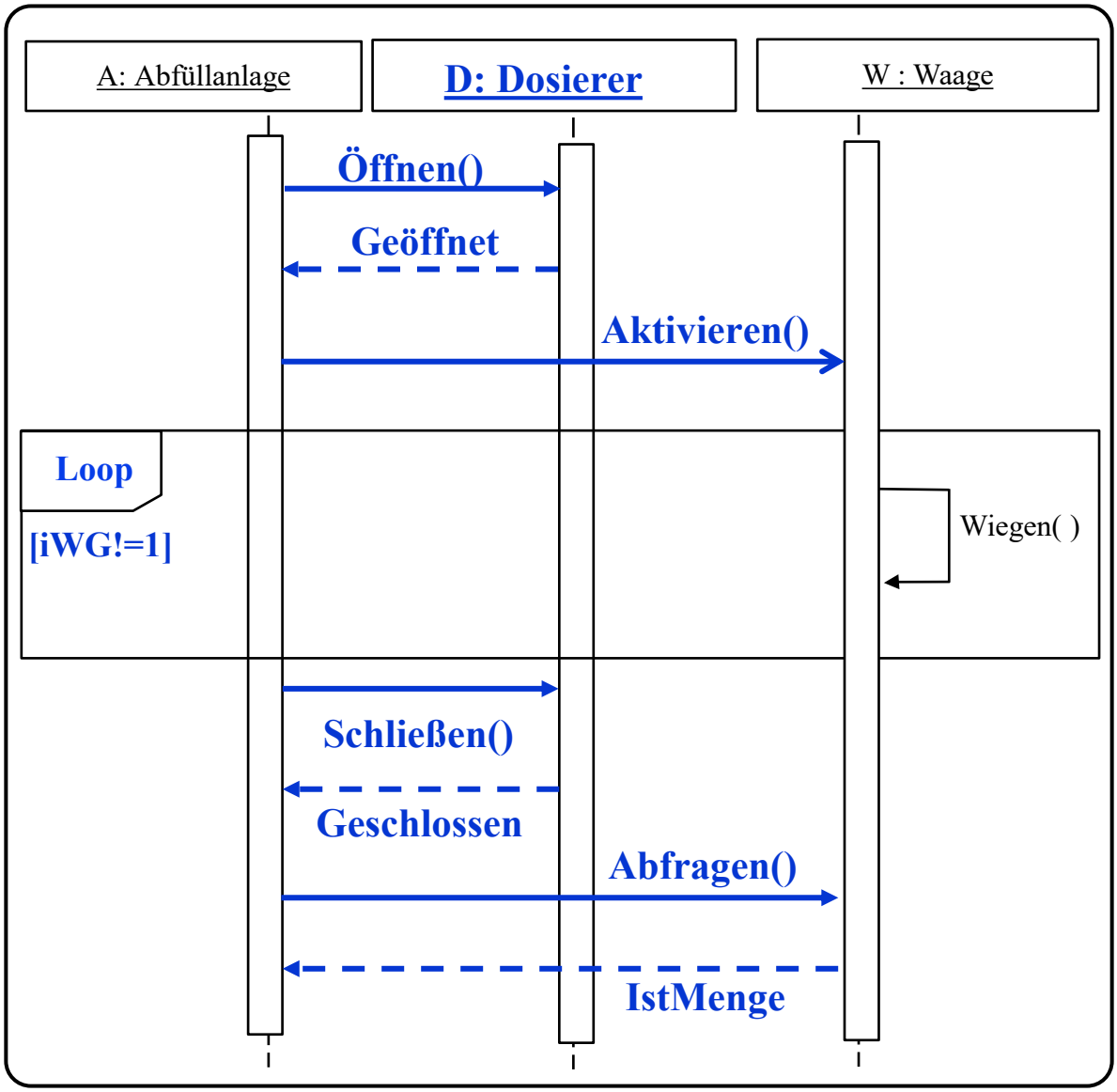
Vervollständigen Sie das untenstehende UML-Use-Case-Diagramm der **`Abfüllanlage`** für Gummibärchen gemäß der oben beschriebenen Anwendungsfälle.



Aufgabe 11: UML-Sequenzdiagramm

Das Abwiegen der Gummibärchen in der **`Abfüllanlage (Objekt A)`** mit Hilfe des **`Dosierers (Objekt D)`** und der **`Waage (Objekt W)`** soll als Sequenzdiagramm modelliert werden. Vervollständigen Sie das folgende Sequenzdiagramm entsprechend der untenstehenden Beschreibung. Achten Sie auf die passenden Pfeilspitzen gemäß der erforderlichen Nachrichtentypen.

- Zunächst öffnet die **`Abfüllanlage`** den **`Dosierer`** (synchrone Nachricht **`Öffnen`**), um Gummibärchen aus dem Lager zu erhalten.
- Sobald der **`Dosierer`** antwortet, dass er **`Geöffnet`** ist, **`Aktiviert`** die **`Abfüllanlage`** die **`Waage`**, erwartet jedoch keine Antwort.
- Die **`Waage`** wiegt so lange, bis das bestellte Wunschgewicht erreicht ist (Variable **`iWG`** nimmt den Wert 1 an, wenn Wunschgewicht erreicht ist).
- Nachdem das Wunschgewicht erreicht wurde, wird der **`Dosierer`** durch die **`Abfüllanlage`** mittels synchroner Kommunikation **`Geschlossen`**.
- Als letztes fragt die **`Abfüllanlage`** mit der synchronen Nachricht **`Abfragen`** von der **`Waage`** den exakt gemessenen Wert (Antwort **`IstMenge`**) ab.

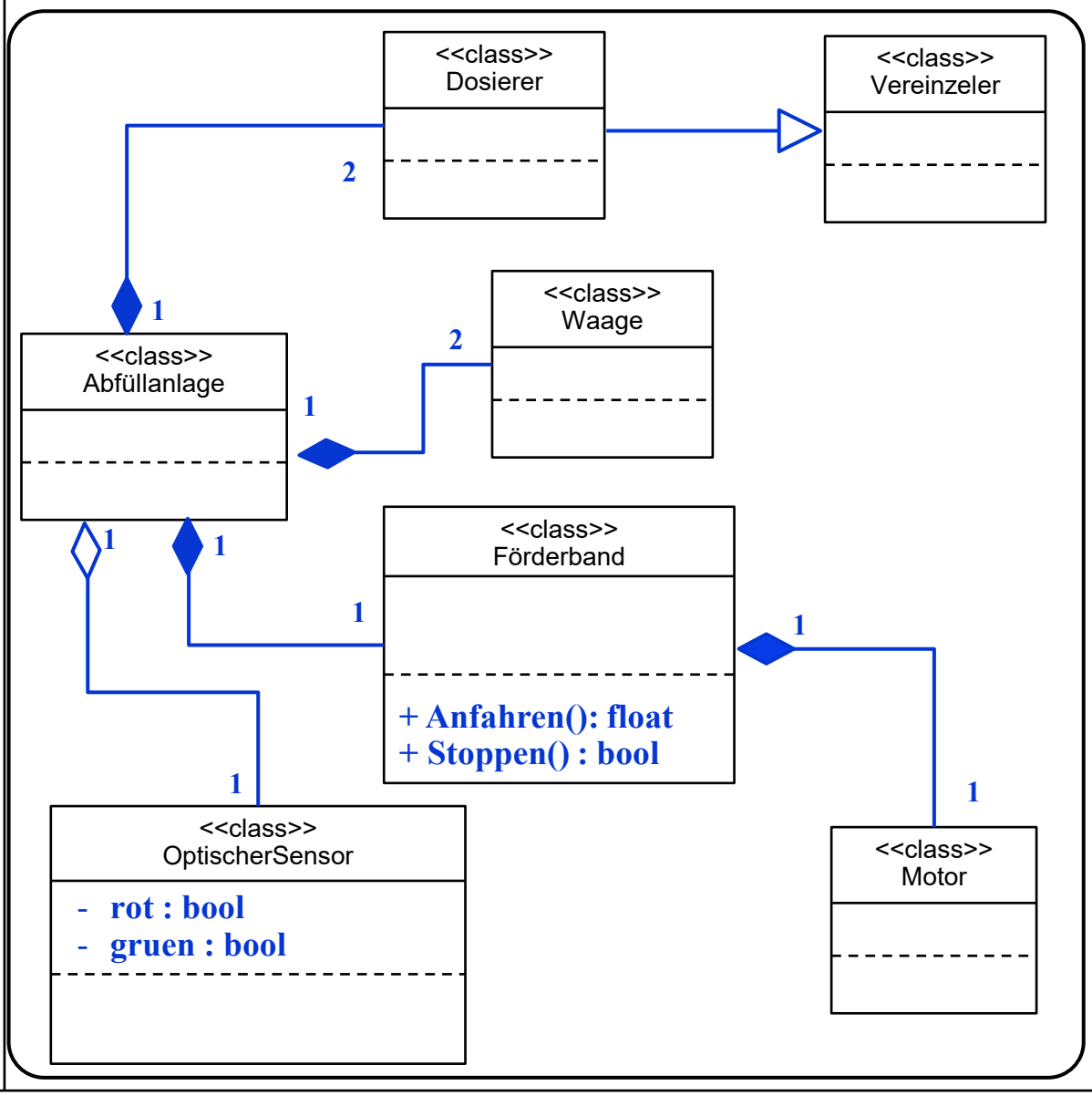


Aufgabe 12: UML-Klassendiagramm

Die **`Abfüllanlage`** besteht immer aus zwei **`Waagen`**, zwei **`Dosierern`** und genau einem **`Förderband`**, sowie optional aus einem optischen Sensor (**`OptischerSensor`**).

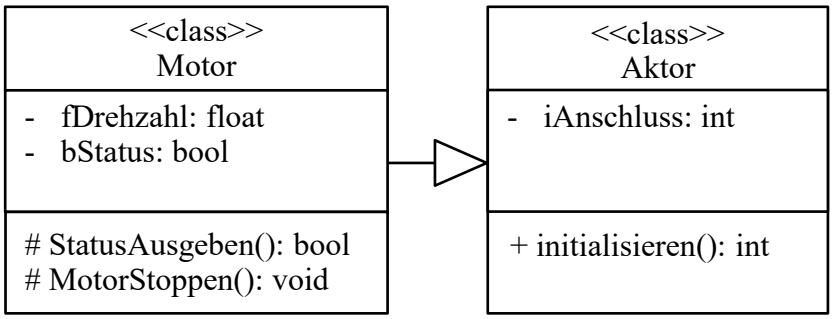
- Der **`Dosierer`** ist eine Spezialisierung der Klasse **`Vereinzeler`**.
- Ein **`Förderband`** hat immer einen **`Motor`** zur Geschwindigkeitsregelung.
- **`OptischerSensor`** verfügt über die beiden privaten, boolschen Attribute **`rot`** und **`gruen`**, die jeweils angeben, ob die Gummibärchen rot bzw. grün sind.
- Mittels der beiden öffentlichen (public) Methoden **`Anfahren()`** und **`Stoppen()`** wird das Förderband gesteuert. **`Anfahren`** startet das Band und gibt die Ist-Geschwindigkeit als float zurück. **`Stoppen`** hält das Förderband an und gibt einen booleschen Wert zurück.

Vervollständigen Sie das untenstehende UML-Klassendiagramm um Klassennamen, Attribute, Methoden und Beziehungen gemäß obiger Beschreibung. Achten Sie hierbei auf die Kardinalitäten.



Aufgabe 13: Überführung eines UML-Klassendiagramms in C++-Code

Im Folgenden wird der Motor am Förderband näher betrachtet. Dazu ist folgendes Klassendiagramm gegeben:



Ergänzen Sie die Klassendeklarationen der zwei dargestellten Klassen **`Motor`** und **`Aktor`** in **C++**.

Hinweise:

- Die Anzahl der Linien im Lösungsfeld ist bei allen Programmieraufgaben unabhängig von der Anzahl an geforderten Codezeilen.
- Alle benötigten Header-Dateien sind bereits eingebunden.

```
class Aktor {

  private:
    int iAnschluss;

  public:
    int initialisieren();

};

class Motor : public Aktor {
  private:
    float fDrehzahl;
    bool bStatus;
  protected:
    bool StatusAusgeben();
    void MotorStoppen();

};
```

Aufgabe 14: Ergänzen Sie das UML-Zustandsdiagramm

Im Folgenden wird der Produktionsablauf für das Abfüllen von roten und grünen Gummibärchen betrachtet (Bild 14.1).

Die Anlage beginnt im Zustand **`Wartend`**. Bei Eintritt werden die Dosierer (**`iDosierer1 = 0, iDosierer2 = 0`**) und das Förderband (**`iFBMotor = 0`**) abgeschaltet. Im Zustand **`Wartend`** wartet die Anlage auf eine neue Bestellung (**`BestWarten()`**). Sobald eine neue Bestellung eingeht (**`iBestErhalten == 1`**), wird das Förderband gestartet (**`iFBMotor = 1`**) und der Zustand **`Wartend`** verlassen.

Je nach bestellter Gummibärchenfarbe, rot (**`farbe == 'R'`**) oder grün (**`farbe == 'G'`**), wird in die Zustände **`Abfüllend-rot`** bzw. **`Abfüllend-grün`** übergegangen. Im Zustand **`Abfüllend-rot`** wird der Dosierer 1 eingeschaltet. Waage 1 misst das Gewicht der roten Gummibärchen, bis das bestellte Wunschgewicht erreicht ist (**`iWG == 1`**). Beim Verlassen des Zustands wird Dosierer 1 abgeschaltet und die Gummibärchen werden zur Prüfung ausgegeben (**`Ausgeben()`**).

Der Zustand **`Abfüllend-grün`** verhält sich analog zum Zustand **`Abfüllend-rot`**, jedoch werden hier Dosierer 2 und Waage 2 statt Dosierer 1 und Waage 1 verwendet.

Im Zustand **`Prüfend`** wird kontrolliert, ob die Farbe der Gummibärchen mit der Bestellung übereinstimmt (Messung dauert 2000ms). Stimmt die Farbe überein (**`iFarbePassend == 1`**), folgt wieder Zustand **`Wartend`**. Stimmt die Farbe nicht überein, wird im Zustand **`Personal Anfordernd`** einmalig eine E-Mail an das Personal verschickt (**`sendeEmail()`**) und das Förderband kontinuierlich gestoppt (**`iFBMotor = 0`**), bis der Fehler manuell behoben wurde .

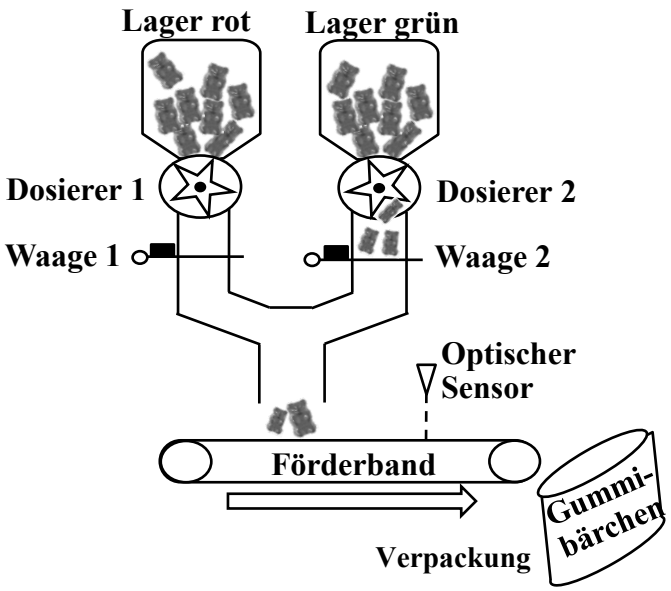


Bild 14.1: *Sensorik und Aktorik der Abfüllanlage für Gummibärchen*

Es ist das in Bild 14.2 gezeigte Zustandsdiagramm mit den Zustandsnummern 1 bis 5 gegeben, welches den beschriebenen Abfüllvorgang abbildet. Geben Sie untenstehend an, wie die durch römische Ziffern gekennzeichneten Lücken gefüllt werden müssen, um der Beschreibung auf vorheriger Seite zu entsprechen.

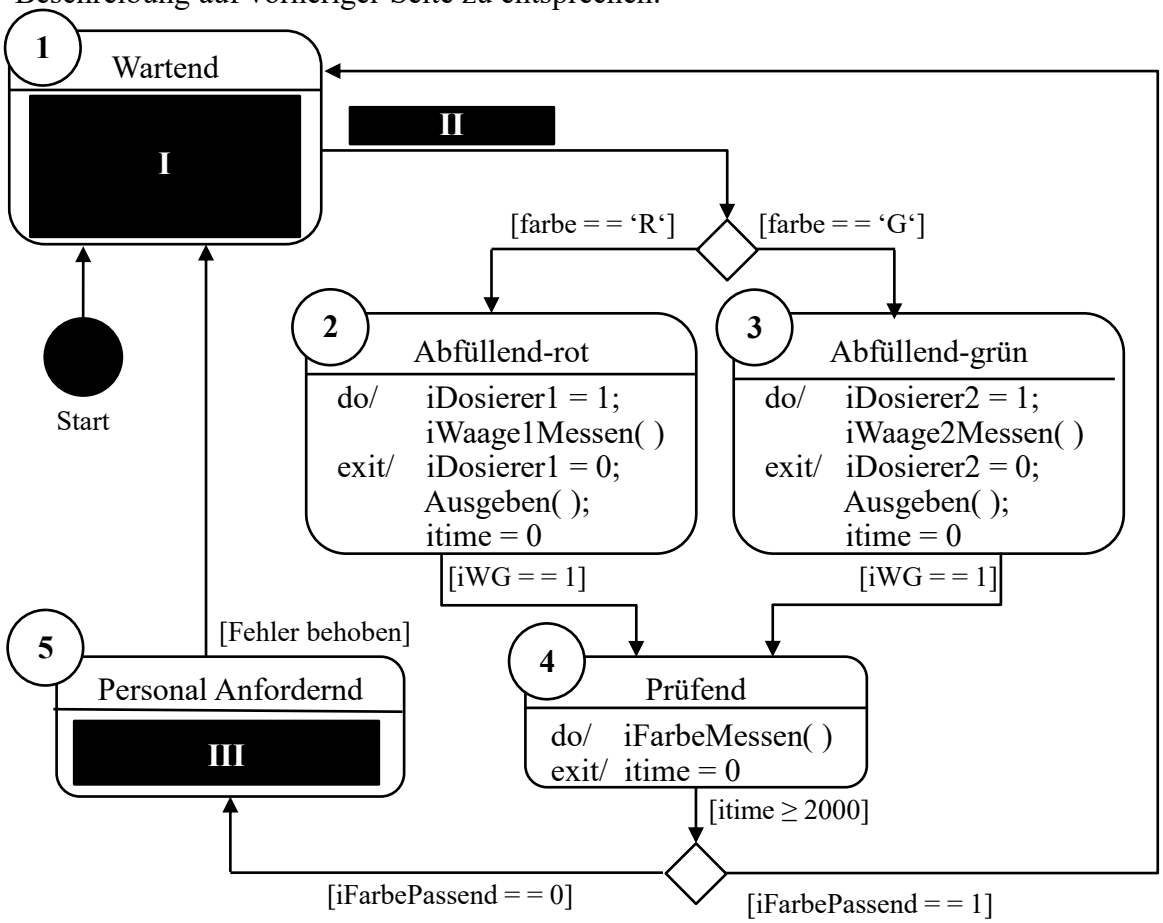


Bild 14.2: *Zustandsdiagramm der Betriebssoftware der Gummibärchen-Abfüllanlage*

I. Modellieren Sie den Zustand 1 **`Wartend`** korrekt aus:

```
entry/iDosierer1 = 0;
      iDosierer2 = 0;
      iFBMotor = 0
do/   BestWarten()
exit/ iFBMotor = 1
```

II. Geben Sie die korrekte Wächterbedingung an:

```
[iBestErhalten == 1]
```

III. Modellieren Sie den Zustand 5 **Personal Anfordernd** korrekt aus:

```
entry /   sendeEmail()
do /      iFBMotor = 0
```

Aufgabe 15: UML-Zustandsdiagramm zu C-Code

Implementieren Sie Teile des in Aufgabe 14 modellierten Zustandsdiagramms in der Programmiersprache C. Nutzen Sie hierfür die in Tabelle 15.1 vorgegebenen Variablen und vorimplementierten Funktionen.

Tabelle 15.1: *Vorgegebene Variablen und vorimplementierte Funktionen*

Typ	Name	Beschreibung
VARIABLEN	`int istate`	Variable für den aktuellen Zustand des Systems mit insgesamt 5 Zuständen {1,2,3,4,5}.
	`int iDosierer1, int iDosierer2`	Variable zur Ansteuerung der Dosierer 1 und Dosierer 2 (`1`: Start; `0`: Stop).
	`int iWG`	Variable zur Überprüfung der Erreichung des Wunschgewichts (`1:` Wunschgewicht erreicht; `0:` andernfalls).
	`unsigned int vplcZeit`	Zählvariable für die aktuelle Zeit der SPS in ms ab Programmstart.
	`unsigned int itime`	Zählvariable für Zeitmessung.
	`int iFarbePassend`	Variable zur Überprüfung der Gummibärchenfarbe (`1:` Gummibärchenfarbe entspricht der bestellten Farbe; `0:` andernfalls).
FUNKTIONEN	`int iWaage1Messen();`	Funktion, die zurückgibt, ob das Wunschgewicht erreicht wurde (`1:` Wunschgewicht erreicht; `0:` andernfalls).
	`void Ausgeben();`	Funktion zur Ausgabe der gewogenen Gummibärchen.
	`int iFarbeMessen();`	Funktion, die zurückgibt, ob die passende Farbe gemessen wurde (`1:` Farbe passend; `0:` andernfalls).

a) Vervollständigen Sie das folgende Programmgerüst in der Programmiersprache C gemäß den Kommentaren im Lösungskästchen. Verwenden Sie hierfür die in Tabelle 15.1 angegebenen Variablennamen und Funktionen, welche im Header **`Abfuellanlage.h`** bereits deklariert und implementiert sind. Verwenden Sie zudem für die Implementierung von Zustand 2: **`Abfüllend-rot`** den in Bild 15.1 gezeigten Ausschnitt des Zustandsdiagramms. Im Zustand **`Abfüllend-rot`** wird die Funktion **`iWaage1Messen()`** aufgerufen und ihr Rückgabewert in der bereits deklarierten Variablen **`iWG`** gespeichert.

Hinweis: Die Platzhalter /*ZUSTAENDE*/ enthalten spezifischen Code für Zustände des Zustandsautomaten und müssen nicht implementiert werden.

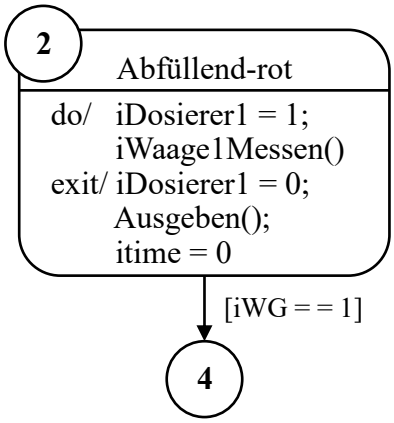


Bild 15.1: *Zustand 2 mit Übergang zu Zustand 4*

```
//Header der Abfuellanlage einbinden
 #include "Abfuellanlage.h"
_____________________________________________________________

// Setzen Sie die Zustandsvariable istate auf Zustand 1
 istate = 1;
_____________________________________________________________

int main() {
// Zyklische Endlosausfuehrung
 while(True)
_____________________________________________________________
  { // Fuellen Sie den Zustandsautomat aus
       switch(istate)
_____________________________________________________________
       {
            case 1:
                 /* ZUSTAENDE */
            case 2:
               iDosierer1 = 1;
               iWG = iWaage1Messen();
               _____________________________________________
               if( iWG == 1    alternativ:   iWG          ){
                 iDosierer1 = 0;
                 Ausgeben();
                 itime = 0;
                 istate = 4;

               }
               break; // Zustand 2 verlassen
       /* ZUSTAENDE */
       }
 }
return 0;}
```

b) Der Zustand 4: **Prüfend** soll im Folgenden implementiert werden (Bild 15.2).

Im Zustand **Prüfend** wird die Funktion **iFarbeMessen()** aufgerufen und ihr Rückgabewert in der bereits deklarierten Variablen **iFarbePassend** gespeichert. Für ein präzises Messergebnis kann der Wert der Variablen **iFarbePassend** erst nach 2000 ms ausgewertet werden. Nach Ablauf der Zeit wird gemäß Bild 15.2 bei passender Farbe in Zustand 1: **Wartend** und bei falscher Farbe in Zustand 5: **Personal Anfordernd** gewechselt.

Hinweise: Verwenden Sie einen Timer, um die Wartezeit von 2000 ms für die Farbmessung zu realisieren. Gehen Sie davon aus, dass der Timer beim erstmaligen Eintritt in den Zustand den Wert **itime=0** hat. Setzen Sie **itime** beim Verlassen des Zustandes **Prüfend** zurück.

Implementieren Sie den Zustand **Prüfend** der Anlage gemäß obiger Beschreibung und Bild 15.2 in der Programmiersprache C. Verwenden Sie zudem die Variablennamen aus Tabelle 15.1.

```
case 4:
  iFarbePassend = iFarbeMessen();

  if(itime==0)
  {
      itime = vplcZeit + 2000;
  }
  else
  {
      if (itime <= vplcZeit)
      {
        itime=0;
              if(iFarbePassend)
              {
                 istate = 1;
              }
              else
              {
                 istate = 5;
              }
       }
  }


break; //Zustand 4 verlassen
```

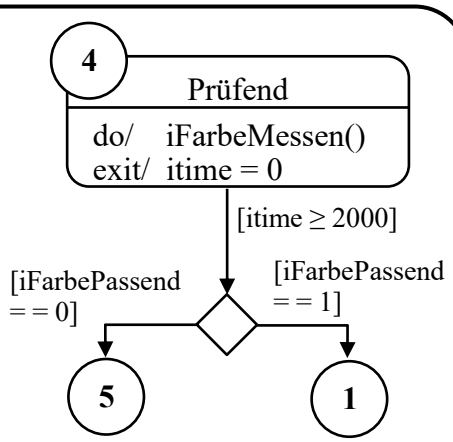


Bild 15.2: *Zustände 4, 5, 1*

alternativ:

```
...
if (itime == 0)
{
   itime = vplcZeit;
}
else
{
   if(vplcZeit-itime>=2000)
   ...
}
```

Aufgabe 16: Grundlagen in C – Datentypen und Kontrollstrukturen

a) Gegeben ist nachfolgende Funktion (**`funcA`**).

```
void funcA (unsigned int a)
{
      if(a==2) { printf("richtig");}
      else { printf("falsch");}
      return;
}
```

Schreiben Sie in der Programmiersprache C eine Funktion **`funcB`**, die bei gleichen Übergabeparametern die gleichen Ausgaben erzeugt wie die Funktion **`funcA`**, hierfür aber **kein** `if, else if, else` verwendet.

Ergänzen Sie das Lösungskästchen.

```
void funcB (unsigned int a)
{
        switch(a) {
            case 2:
                printf("richtig");
                break;
            default:
                printf("falsch");
                break;      Hint: Break optional
        }

    return;
}
```

b) Deklarieren Sie in der Programmiersprache C eine Variable **`B`** vom Typ float und initialisieren Sie diese mit dem Wert **`4.4`**. Deklarieren Sie einen Pointer **`pB`**, welcher auf die Variable **`B`** verweist. Setzen Sie anschließend den Wert der Variablen **`B`** auf **`2.1`**, indem Sie den Pointer **`pB`** verwenden.

```
float B = 4.4;
float *pB = &B;
*pB = 2.1;
```

Aufgabe 17: Algorithmen

Es soll eine neue Abfüllanlage für Gummibärchen entworfen werden. Dazu wird die Anordnung der einzelnen Anlagenbestandteile geplant und für eine schnelle Abfüllung optimiert. Für die Optimierung des Anlagenlayouts soll eine Funktion (**cityblock**) implementiert werden, welche den Cityblock-Abstand **d(a,b)** zwischen zwei Anlagenteilen a und b berechnet. Der Cityblock-Abstand **d(a,b)** ist die Summe der absoluten Differenz der drei Einzelkoordinaten a_j und b_j der Punkte a und b mit $j \in \{x,y,z\}$.

Formel für Cityblock-Abstand: $\mathbf{d}(a,b) = \sum_{j \in \{x,y,z\}} |a_j - b_j|$

Um den Absolutbetrag |x| einer reellen Zahl x zu berechnen, wurde bereits eine Funktion **absolut** implementiert, welche Sie verwenden sollen:

```
float absolut (float x){
      if(x < 0) { return -x; }
      else { return x; }
}
```

Die Funktion (**cityblock**) hat als Rückgabewert den Cityblock-Abstand vom Typ float und bekommt als Übergabeparameter übergeben:

- einen Funktionszeiger (**betrag**), um beim Aufruf der Funktion **cityblock** die Funktion **absolut** verwenden zu können
- einen Zeiger (**pa**), welcher auf ein eindimensionales Array (**a**) mit 3 Einzelkoordinaten (x,y,z) vom Typ float verweist
- einen Zeiger (**pb**), welcher auf ein eindimensionales Array (**b**) mit 3 Einzelkoordinaten (x,y,z) vom Typ float verweist

Implementieren Sie die Funktion **cityblock** in der Programmiersprache C, indem Sie das Lösungskästchen ergänzen.

```
float cityblock (float (*betrag)(float), float *pa, float *pb)
{
        float sum = 0.0;
        for (int i = 0; i<3; i++)
        {
            sum += betrag(*(pa+i) - *(pb+i));
        }
    return sum;
}
```

Alternativen:

(*betrag) (*(pa+i) - *(pb+i));

(*betrag) (pa[i] – pb[i]);

betrag(pa[i]-pb[i]);

Aufgabe 18: Datenstrukturen

In einer Verpackungsanlage werden die Gummibärchen in Beutel abgefüllt. Implementieren Sie eine einfache Datenbankfunktionalität in Form einer **einfach verketteten Liste mit Listenkopf**, in welcher die abgefüllten Beutel gespeichert werden. Jeder Beutel wird genau einem Listenelement zugeordnet.

Im Listenkopf vom Typ **`ANLAGE`** sollen gespeichert werden:

- Der Name der Abfüllanlage (**`name`**) als Zeichenkette mit exakt 5 Buchstaben. Berücksichtigen Sie dabei ein abschließendes Nullzeichen bei Zeichenketten.
- Ein Zeiger (**`pfirst`**) auf das erste Listenelement vom Typ **`BEUTEL`**.

Im Listenelement vom Typ **`BEUTEL`** sollen gespeichert werden:

- Nutzdaten (**`eigenschaften`**) vom Typ `DATA`.
- Ein Zeiger (**`pnext`**) auf das nachfolgende Listenelement.

In den Nutzdaten vom Typ **`DATA`** sollen gespeichert werden:

- Anzahl der enthaltenen Gummibärchen (**`menge`**) als vorzeichenlose Ganzzahl mit dem Wertebereich 0 … 255. Achten Sie auf Speichereffizienz. Hinweis: Auf der verwendeten Architektur hat ein unsigned short 16bit und den Wertebereich 0…65535
- Das exakte Füllgewicht der enthaltenen Gummibärchen (**`gewicht`**) als Gleitkommazahl mit kleinstmöglicher Speicherbelegung in Gramm.

a) Implementieren Sie die Nutzdaten vom Typ **`DATA`**, ein Listenelement vom Typ **`BEUTEL`** sowie den Listenkopf vom Typ **`ANLAGE`**, indem Sie die Lösungskästchen in der Programmiersprache C ausfüllen.

```
typedef struct {
        unsigned char menge;      alt.: uint8_t statt char
        float gewicht;
} DATA;
```

```
typedef struct beutel {
        DATA eigenschaften;
        struct beutel *pnext;

} BEUTEL;
```

```
typedef struct {
    char name[6];
    BEUTEL *pfirst;
} ANLAGE;
```

Aus technischen Gründen kann das Abfüllgewicht (**gewicht**) der Gummibärchen in jedem Beutel schwanken. In einer Funktion (**statistik**) sollen berechnet werden:

- das Abfüllgewicht des schwersten Gummibärchenbeutels in der Datenbank
- die Gesamtanzahl (**beutelzahl**) an Gummibärchenbeuteln in der Datenbank

Die Funktion **statistik** hat als Rückgabewert die Gesamtanzahl (**beutelzahl**) der Gummibärchenbeutel und bekommt folgende Übergabeparameter übergeben:

- einen Zeiger (**start**) vom Typ **ANLAGE** auf den Listenkopf
- einen Zeiger (**maxgewicht**) vom Typ float, welcher auf die Variable verweist, in der das Abfüllgewicht des schwersten Gummibärchenbeutels abgespeichert werden soll.

b) Implementieren Sie die Funktion **statistik** in der Programmiersprache C, indem Sie das Lösungskästchen ergänzen. Beachten Sie dabei, dass der Zeiger **pnext** des letzten Listenelementes auf **NULL** zeigt und die Liste nicht leer ist.

```
int statistik (ANLAGE *start, float* maxgewicht)
{
// Zeiger auf das erste Listenelement vom Typ BEUTEL
    BEUTEL* tmp = start->pfirst;

  int beutelzahl = 0;
  *maxgewicht = 0;
  while(tmp != NULL)
  {
      if (tmp->eigenschaften.gewicht > *maxgewicht)
      {
          *maxgewicht = tmp->eigenschaften.gewicht;
      }
      tmp = tmp->pnext;
      beutelzahl++;

   }




    return beutelzahl;
}
```