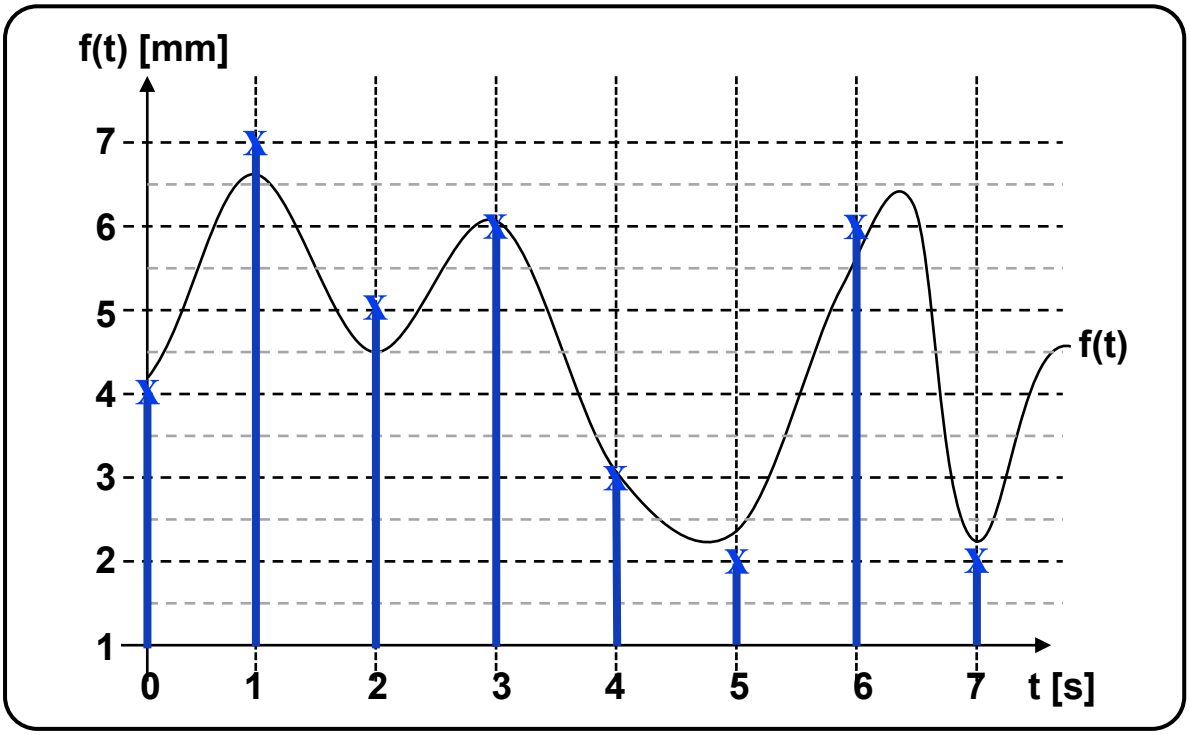
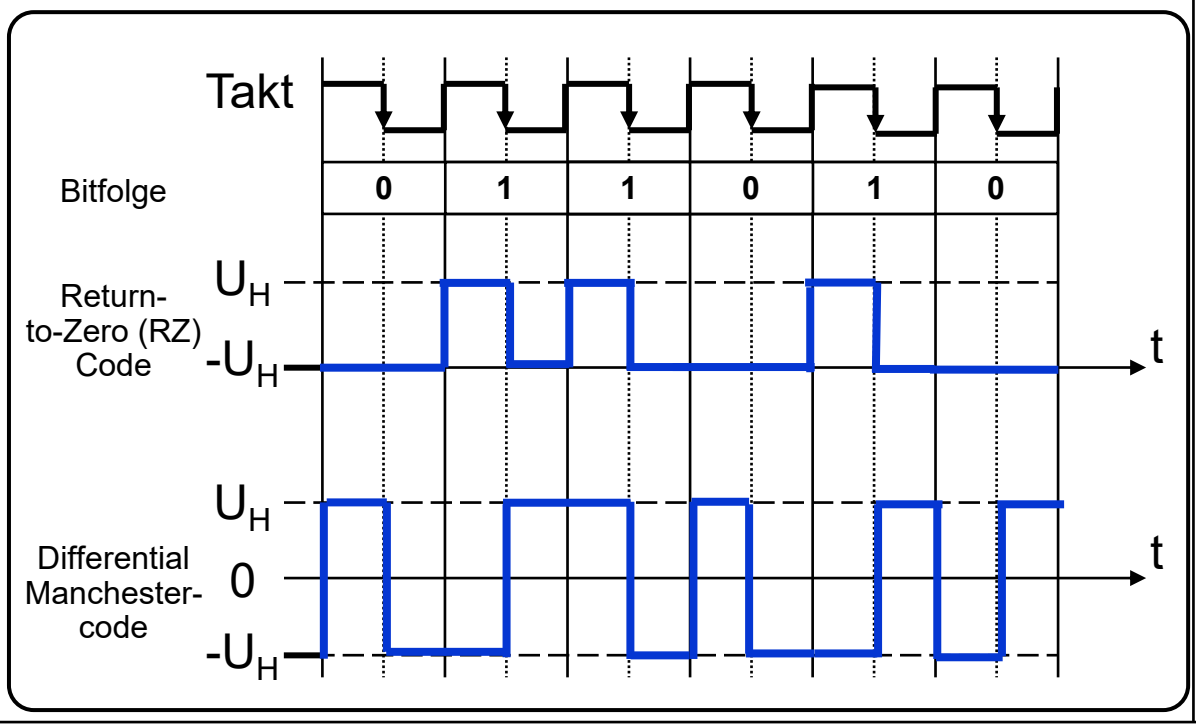
Aufgabe 1: Grundlagen der Informationstechnik und Digitaltechnik

a) Gegeben ist das dargestellte, wert- und zeitkontinuierliche Signal f(t). Zeichnen Sie den wertdiskreten und zeitdiskreten Signalverlauf von f(t) für t ∈ [0;7] in das Schaubild ein. Die Diskretisierung erfolgt jede Sekunde [s] auf ganzzahlige mm, wobei Werte mit der ersten Dezimalstelle >=5 aufgerundet und sonst abgerundet werden (z.B. 3.5 → 4).



b) Sie versenden die **Bitfolge `011010`** auf einem seriellen Bussystem. Zeichnen Sie den resultierenden Leitungscode als Return-To-Zero (RZ) Code und im Differential Manchestercode. Für beide Codes liegt **zu Beginn $-U_H$** an.



Aufgabe 2: IEEE 754 Gleitkommadarstellung und Zahlensysteme

a) Rechnen Sie die Dezimalzahl $(+17{,}125)_{10}$ 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 4 Bit	Mantisse M 7 Bit

Vorzeichenbit

0

Dezimalzahl $(17{,}125)_{10}$ als Binärzahl

10001,001

Bias als Dezimalzahl

$$B = 2^{(x-1)} - 1 = 2^{(4-1)} - 1 = (7)_{10}$$

Exponent als Dezimalzahl

$$e = (4)_{10}$$

Biased Exponent als Dualzahl

$$E = e + B = 7 + 4 = (11)_{10} = (1011)_2$$

Vollständige Gleitkommazahl (nach obigem Schema)

0	**1011**	**0001001**
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 $(AE)_{16} = (\underline{10101110})_2 = (\underline{256})_8$

2 $(35)_6 = (\underline{23})_{10} = (\underline{43})_5$

Aufgabe 3: Logische Schaltungen und Schaltbilder

a) Vervollständigen Sie die Wahrheitstabelle (Tab. 3.1) für die gegebene Schaltung (Bild 3.1).

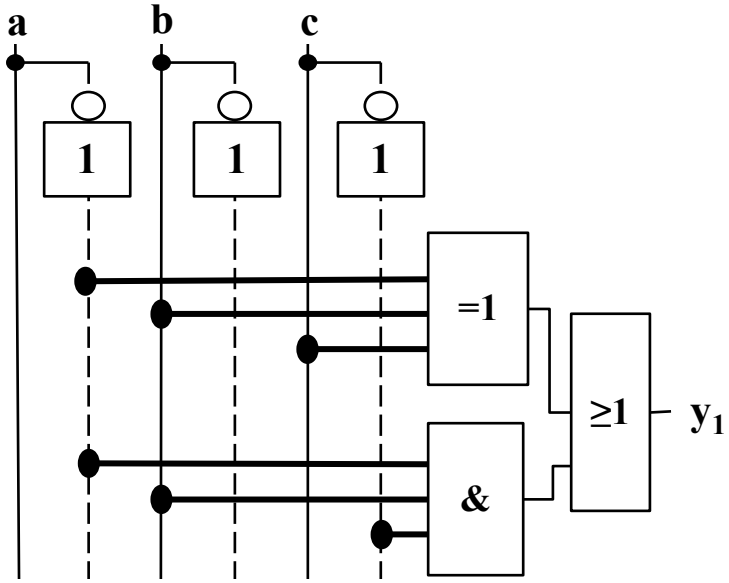


Bild 3.1: *Schaltung*

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

Tab. 3.1: *Wahrheitstabelle zu Bild 3.1*

b) Gegeben ist folgende Wahrheitstabelle (Tab. 3.2). Stellen Sie die zugehörige Disjunktive Normalform (DNF)-Gleichung auf.

Hinweis: Die Schreibweise a ∧ b können Sie mit ab abkürzen.

y1= ($\bar{a}$ ∧ $\bar{b}$ ∧ c) ∨ ($\bar{a}$ ∧ b ∧ $\bar{c}$) ∨ (a ∧ b ∧ c)
Alt.:
y1= ($\bar{a}$ $\bar{b}$ c) ∨ ($\bar{a}$ b $\bar{c}$) ∨ (a b c)

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

Tab. 3.2: *Wahrheitstabelle*

Aufgabe 4: Flip-Flops

Gegeben ist die folgende Master-Slave (Primary-Secondary)-Flip-Flop-Schaltung:

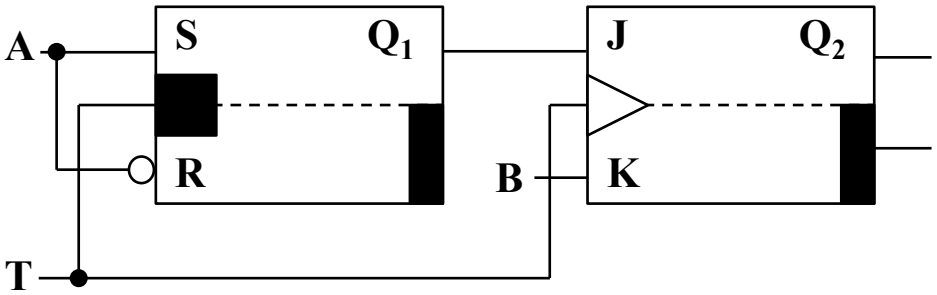
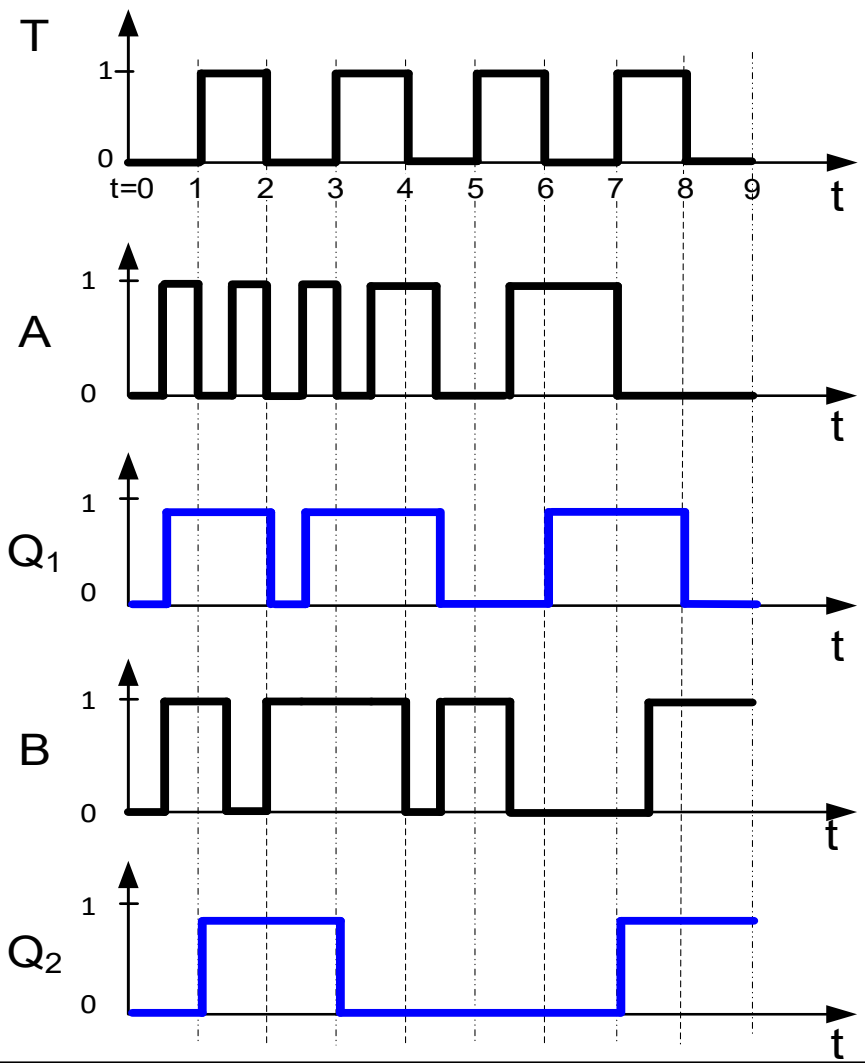


Bild 4.1: *Master-Slave (Primary-Secondary)-Flip-Flop*

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

Analysieren Sie die Schaltung, indem Sie für die gegebenen Eingangssignale A, B und T die zeitlichen Verläufe für Q1 und Q2 in die vorgegebenen Koordinatensysteme für die Zeitspanne 0-9s eintragen.

Hinweis: Signallaufzeiten können bei der Analyse vernachlässigt werden.



Aufgabe 5: Automaten

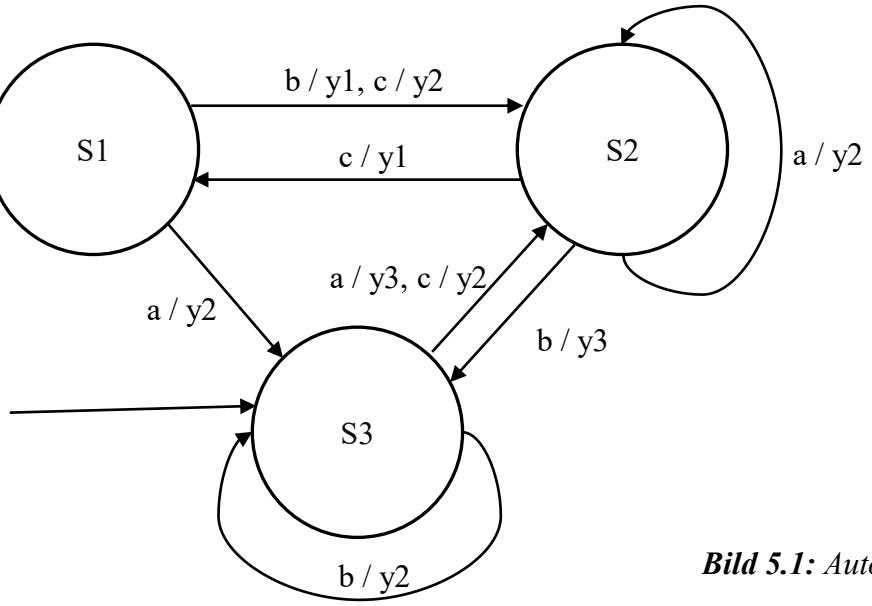


Bild 5.1: *Automat*

a) Ist der in Bild 5.1 gezeigte Automat ein Mealy- oder Moore-Automat?

Mealy (-Automat)

b) Was ist der Startzustand des in Bild 5.1 gezeigten Automaten?

S3

c) Vervollständigen Sie die Übergangstabelle für den in Bild 5.1 abgebildeten Automaten.

T	S1	S2	S3
a	S3, y2	S2, y2	S2, y3
b	S2, y1	S3, y3	S3, y2
c	S2, y2	S1, y1	S2, y2

d) Welche Ausgabe erhalten Sie ausgehend vom Startzustand für die Eingabesequenz „a, b, b, a, c“ in den Automaten (Bild 5.1)?
In welchem Zustand befindet sich der Automat nach dieser Eingabe?

Ausgabe: **y3, y3, y2, y3, y1**

Zustand nach Eingabe: **S1** (da S3 -a-> S2 -b-> S3 -b-> S3 -a-> S2 -c-> **S1**)

Aufgabe 6: MMIX – Assembler-Code

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

Tab. 6.1: MMIX-Code-Tabelle

Algorithmus:

$$y = \left(\frac{5 \bullet (a - 13)}{b}\right)^2 + c$$

Registerspeicher		
Adresse	Wert vor Befehlsausführung	Kommentar
…	…	…
$0x91	0x00 00 00 00 00 00 FE 01	Zwischen-ergebnis
$0x92	0x00 00 00 00 00 00 52 AB	Variable a
$0x93	0x00 00 00 00 00 00 53 10	Variable b
$0x94	0x00 00 00 00 00 00 AE B2	Variable c
$0x95	0x00 00 00 00 00 00 FF 00	Variable y
$0x96	0x00 00 00 00 00 00 51 0A	Nicht veränderbar
…	…	…

Tab. 6.2: Registerspeicher

Im Registerspeicher eines MMIX-Rechners befinden sich zu Beginn die in Tab. 6.2 gegebenen Werte. In der zusätzlichen Spalte Kommentar ist 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 Tab. 6.1 **umrahmten Befehlsbereiche**. **Speichern Sie die Zwischenergebnisse** nach jedem Befehl des Algorithmus in der Registerzelle mit dem Kommentar **Zwischenergebnis**. Das Endergebnis soll in der Registerzelle mit dem Kommentar **Variable y** gespeichert werden. Übersetzen Sie die Operationen in **Assembler-Code mit insgesamt maximal 5 Anweisungen**.

1	SUBI $0x91, $0x92, 0x0D	Alt.: SUBI $0x91, $0x92, 0x0D
2	MULI $0x91, $0x91, 0x05	DIV $0x91, $0x91, $0x93
3	DIV $0x91, $0x91, $0x93	MULI $0x91, $0x91, 0x05
4	MUL $0x91, $0x91, $0x91	MUL $0x91, $0x91, $0x91
5	ADD $0x95, $0x$91, $0x94	ADD $0x95, $0x$91, $0x94

b) Nehmen Sie an, der Inhalt des Datenspeichers entspricht nun dem Zustand in Tab. 6.3. Laden Sie ein Wyde ab Speicherstelle M[0x0…510D] in die Variable c im Registerspeicher (Tab. 6.2). Geben Sie **genau einen**, hierfür notwendige Befehl in Assembler-Code an. Wie lautet der 64-Bit-Wert der Variablen c nach dem Ladevorgang in Hexadezimalschreibweise?
Hinweis: Für diese Teilaufgabe gilt die Big-Endian Adressierung.

Assembler-Befehl:

LDWI $0x94, $0x96, 0x03

Alt.:
LDWI $0x94, $0x96, 0x02

Wert (64 Bit):

0x 00 00 00 00 00 00 BA 98

Datenspeicher	
Adresse	Wert
…	…
M[0x00 … 51 07]	0x54
M[0x00 … 51 08]	0x32
M[0x00 … 51 09]	0x10
M[0x00 … 51 0A]	0xFE
M[0x00 … 51 0B]	0xDC
M[0x00 … 51 0C]	0xBA
M[0x00 … 51 0D]	0x98
M[0x00 … 51 0E]	0x76
M[0x00 … 51 0F]	0x54
M[0x00 … 51 10]	0x32
M[0x00 … 51 11]	0x10
M[0x00 … 51 12]	0x54
M[0x00 … 51 13]	0x32
…	…

Tab. 6.3: *Datenspeicher*

c) Übersetzen Sie den folgenden Befehl aus Maschinensprache (binär) in Maschinensprache (Hexadezimal) und in Assembler-Code.
Hinweis: Nutzen Sie die Informationen aus Tab. 6.1.

Maschinensprache (Binär): 1010 0000 1010 1010 0000 1000 1111 0001

Maschinensprache (Hexadezimal): 0x A 0 A A 0 8 F 1

Assemblersprache: STB $0xAA, $0x08, $0xF1

Aufgabe 7: Round-Robin-Scheduling mit Semaphoren

Gegeben seien der folgende Soll-Verlauf der vier Tasks A, B, C und D (Bild 7.1) sowie die Anordnung der Semaphor-Operationen am Anfang und am Ende der Tasks (Tab. 7.1).

Die vier Tasks werden nach einem Round-Robin-Verfahren mit festen Zeitschlitzen von 2 Sekunden eingeplant. Treten zwei Tasks im selben Zeitschlitz auf, werden diese zuerst nach dem Prinzip First-In-First-Out (FIFO) und anschließend alphabetisch gescheduled. Soll gemäß Scheduling-Verfahren einem Task die CPU zugeteilt werden, der aber aufgrund fehlender Semaphore nicht starten kann, wird der Task stattdessen übersprungen und hinten auf die Round-Robin-Scheibe eingereiht.

Tragen Sie für jeden Zeitraum von 1s in t ∈ [0;10] ein, welcher Task auf der CPU läuft. Läuft kein Task auf der CPU, tragen Sie „-“ ein. Gehen Sie von einer Anfangsbelegung der Semaphor-Variablen gemäß Tab. 7.2 aus. Geben Sie die Werte der Semaphor-Variablen an den drei eingezeichneten Zeitpunkten (t = 2.5; t = 5.5; t = 10s) an.

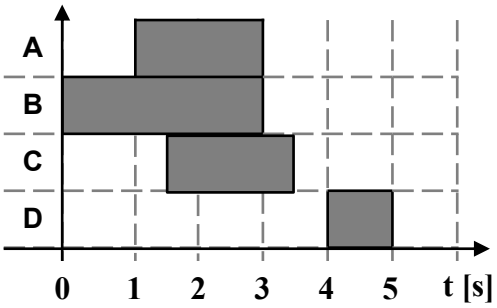


Bild 7.1: *Einplanung / Soll-Verlauf der Tasks*

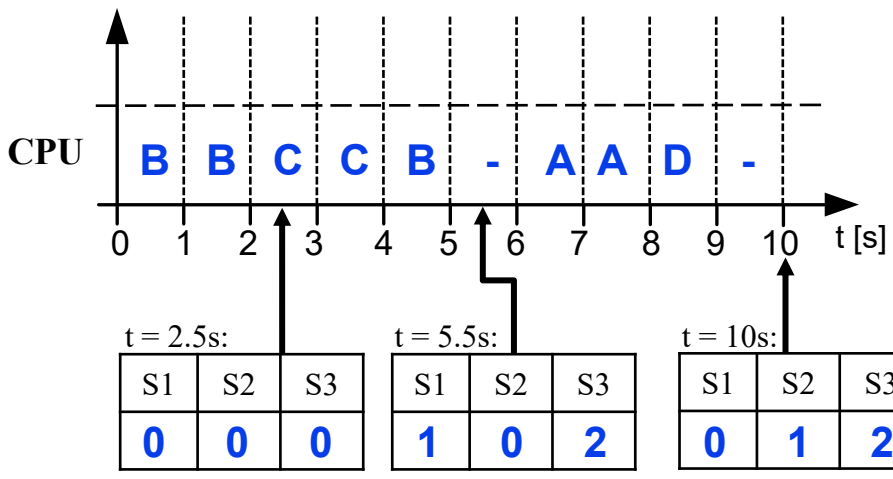
Task	A	B	C	D
t ↓	P(S1) … V(S2)	P(S2) P(S3) … V(S1)	… V(S3) V(S3)	…

Tab. 7.1: *Semaphor-Operationen*

S1	S2	S3
0	1	1

Tab. 7.2: *Initialwerte der Semaphor-Variablen*

Tasks auf der CPU:



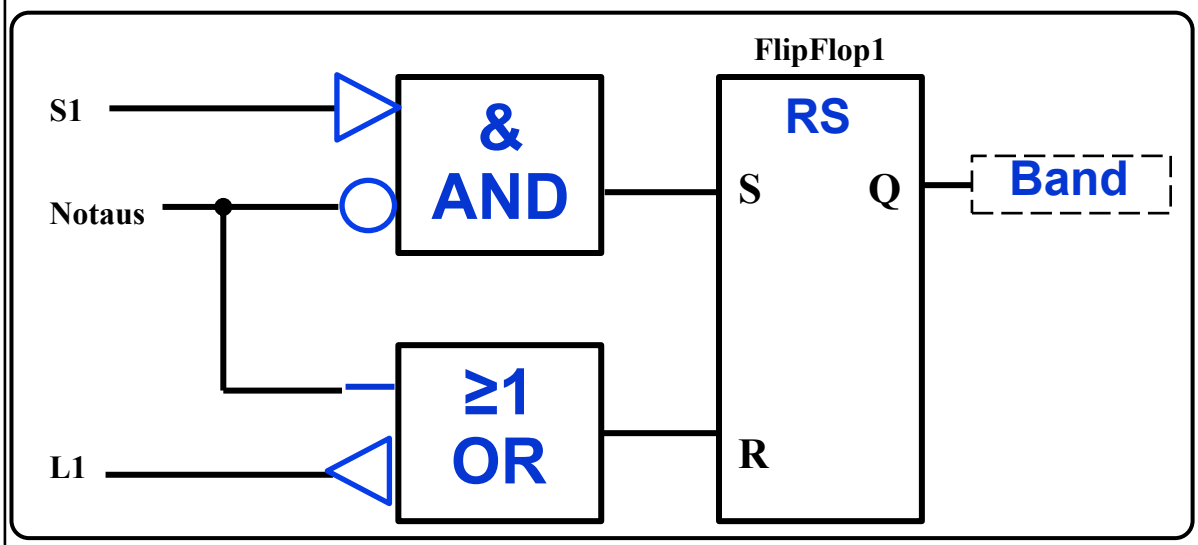
Aufgabe 8: IEC 61131-3 Funktionsbausteinsprache und Ablaufsprache

a) Ergänzen Sie das untenstehende Programm für die Steuerung eines Förderbands in einem Produktionsprozess in IEC 61131-3 Funktionsbausteinsprache (FBS).

- Das Förderband startet (`Band=1`), wenn der Sensor**taster (S1)** betätigt wird und der Notaus inaktiv ist (`Notaus=0`). Verhindern Sie, dass ein dauerhaft gedrückter Sensortaster `S1` das Förderband immer wieder startet.
- Das Förderband stoppt (`Band=0`), falls der Notaus aktiv ist (`Notaus=1`) oder sobald die Lichtschranke `L1` unterbrochen wird (`L1` wird `0`). Eine dauerhaft unterbrochene Lichtschranke stoppt das Förderband nicht. Im Zweifel stoppt das Förderband immer.

Hinweise:

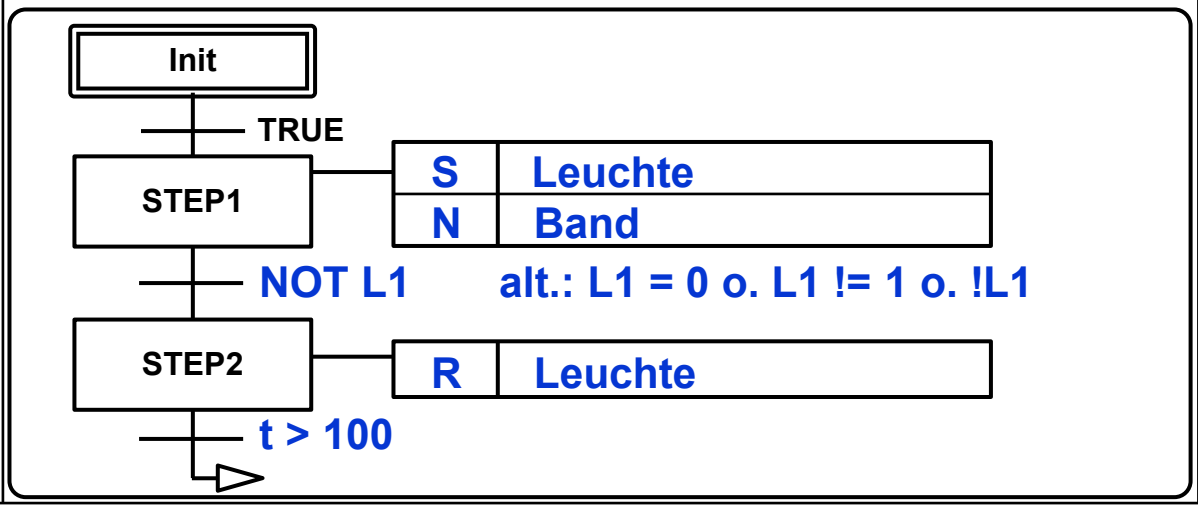
- Signalverzögerungen im System sind zu vernachlässigen.
- Verwenden Sie **keine** Schaltglieder außer den in der Vorlage bereits vorhandenen.
- **Ergänzen Sie Negationen, Flankenerkennung und Flipflopart falls notwendig**.



b) Stellen Sie das folgende Verhalten in IEC 61131-3 Ablaufsprache (AS) dar.

- In STEP1 wird die Warnleuchte eingeschaltet (`Leuchte=1`). Das Förderband läuft nur während STEP1 (`Band=1`). STEP1 wird verlassen, wenn die Lichtschranke `L1` unterbrochen ist (`L1` ist `0`).
- In STEP2 wird die Warnleuchte wieder zurückgesetzt (`Leuchte=0`). STEP2 wird verlassen, sobald die Zeitvariable `t` größer 100 (ms) ist.

Ergänzen Sie die Transitionsbedingungen sowie die Aktionen in den jeweiligen Schritten.



Aufgabe 9: Echtzeitprogrammiersprache PEARL

Vervollständigen Sie den untenstehenden Codeausschnitt in PEARL gemäß den Kommentaren über den Lücken.

```
// Definieren Sie die Unterbrechung „L1“ für eine Lichtschranke.
      SPC      L1            INTERRUPT                    ;
// Definieren Sie den Task „foerdern“ mit Priorität 2.
foerdern:      TASK   PRIORITY 2                          ;
// Der Task „starten“ wird aktiviert.
               ACTIVATE starten                           ;
// Von nun an soll auf die Unterbrechung L1 reagiert werden.
               ENABLE   L1                                ;
// Wenn die Unterbrechung L1 auftritt, wird der Task „starten“
blockiert.
               WHEN   L1
               SUSPEND starten                            ;
// Nach 20 Minuten wird der Task „starten“ beendet.
               AFTER 00 HRS 20 MIN 00 SEC
               TERMINATE starten                          ;
END;
```

Aufgabe 10: UML-Use-Case-Diagramm

Mit der im Bild 10.1 gezeigten automatischen **`Saatgutmaschine`** können Saatgüter in gleichmäßigen Abständen entlang einer definierten Pflanzenreihe auf einem Feld ausgesät werden.

Vor dem Betrieb der Maschine muss der **`Benutzer`** über das Bedienelement die **`Maschine initialisieren`**.

Bei der Initialisierung muss der Benutzer den für die aktuelle Aufgabe genutzten **`Behälter auswählen`**. Außerdem ist Eingabe des Abstands zwischen den Pflanzpositionen (**`Pflanzabstand eingeben`**) notwendig.

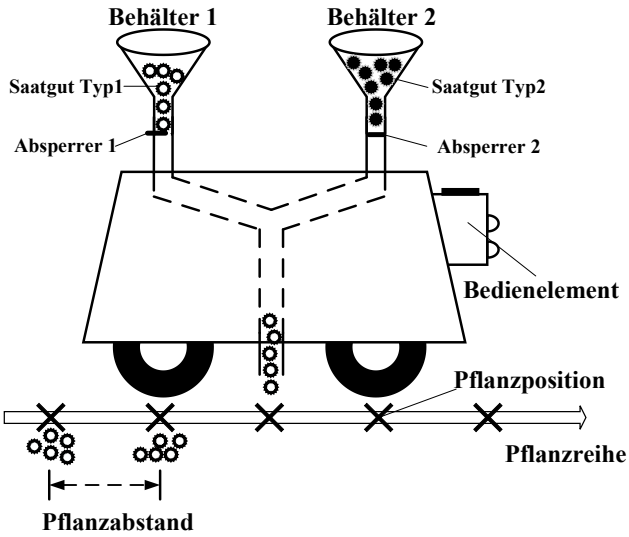
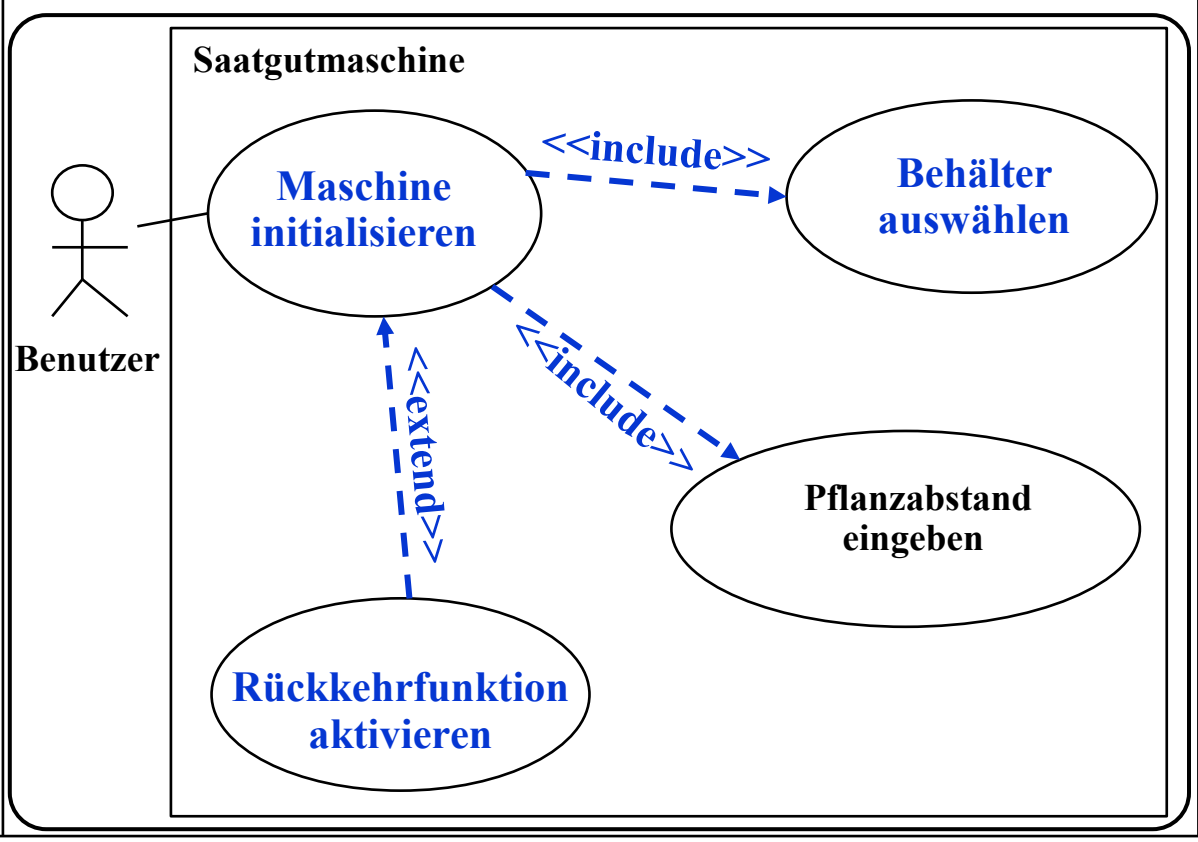


Bild 10.1: *Schematischer Aufbau der Saatgutmaschine*

Die Saatgutmaschine ist mit einer automatischen Rückkehrfunktion ausgestattet, die der **`Benutzer`** bei Bedarf während der Initialisierung aktivieren kann (**`Rückkehrfunktion aktivieren`**). Auf diese Weise kehrt die Maschine nach der Ausführung der aktuellen Aufgabe selbstständig an die Ausgangsposition zurück.

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

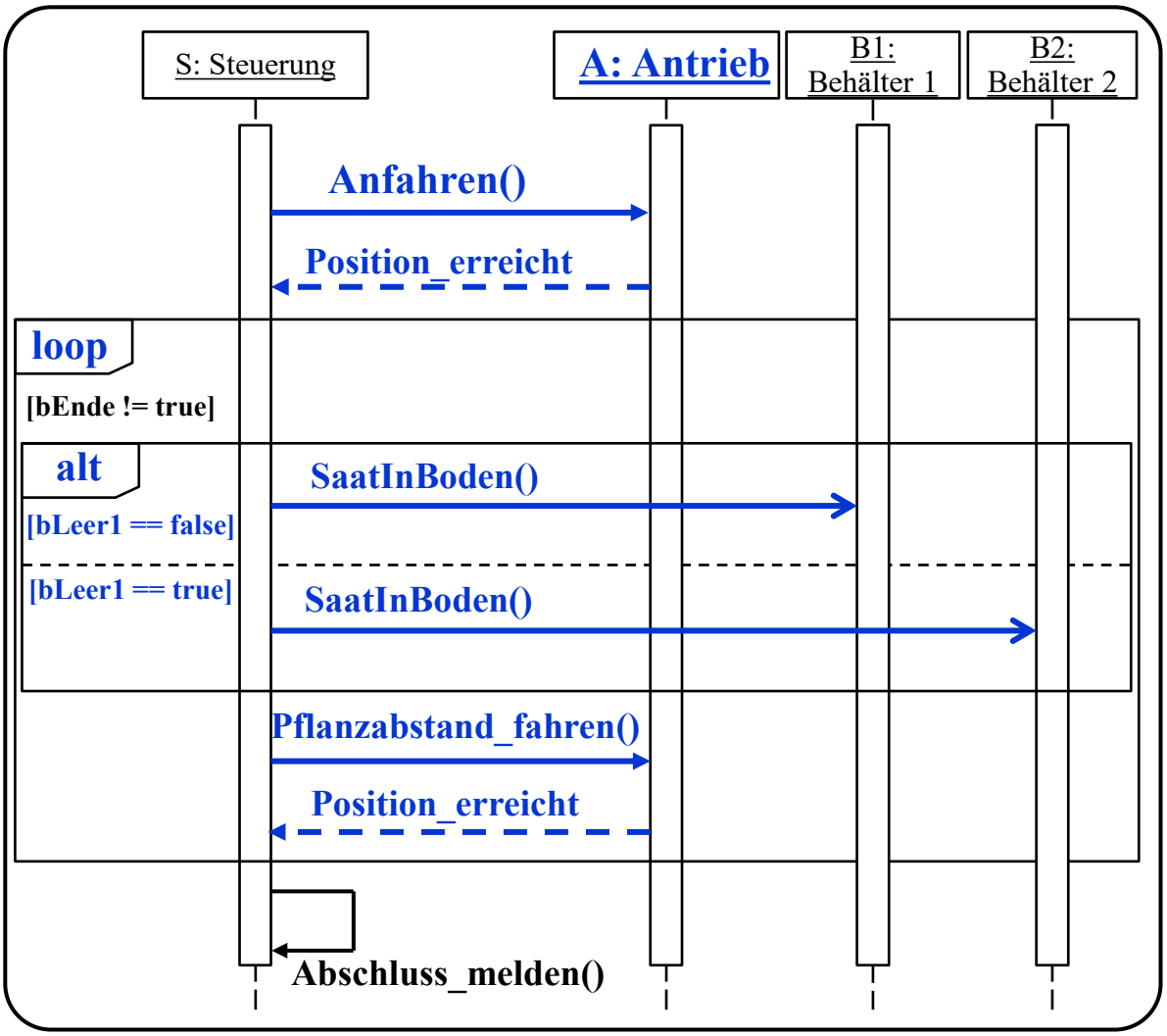


Aufgabe 11: UML-Sequenzdiagramm

Der Aussaatprozess einer Pflanzenreihe mit Hilfe der **`Steuerung (Objekt S)`**, dem **`Antrieb (Objekt A)`** und den beiden (Saatgut-)**`Behältern (Objekt B1`** und **`B2)`** soll als Sequenzdiagramm modelliert werden. In dieser Aufgabe sind die Saatguttypen in Behälter 1 und Behälter 2 identisch.

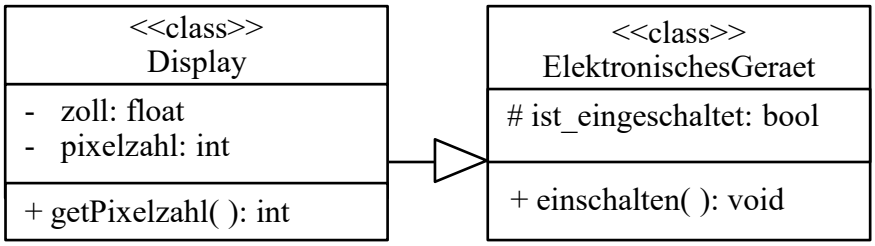
Vervollständigen Sie das folgende Sequenzdiagramm entsprechend der untenstehenden Beschreibung. Achten Sie auf die passenden Pfeilspitzen gemäß der erforderlichen Nachrichtentypen.

- Zunächst fordert die **`Steuerung`** den **`Antrieb`** auf, die erste Pflanzposition anzufahren (**`Anfahren`**) und wartet auf die Antwort „**`Position_erreicht`**“ des Antriebs.
- Danach beginnt die Maschine mit der **zyklischen** Arbeit. Wenn der **`Behälter 1`** nicht leer ist (d.h. **`bLeer1`** ist **`false`**), wird der Behälter 1 von der **`Steuerung`** geöffnet, sodass das Saatgut in den Boden gelangen kann (**`SaatInBoden`**). Wenn **`Behälter 1`** leer ist, wird stattdessen der **`Behälter 2`** geöffnet. Die Steuerung erwartet keine Antwort.
- Danach schickt die Steuerung dem **`Antrieb`** eine synchrone Nachricht, um zur nächsten Pflanzposition zu fahren (**`Pflanzabstand_fahren`**).
- Die Steuerung meldet den Abschluss (**`Abschluss_melden`**) der Aufgabe, wenn das Anlegen der Pflanzenreihe abgeschlossen ist.



Aufgabe 12: Überführung eines UML-Klassendiagramms in C++-Code

Als Teil des Bedienelements wird im Folgenden das Display der Saatgutmaschine näher betrachtet. Dazu ist folgendes Klassendiagramm gegeben:



Ergänzen Sie in **C++** die Klassendeklarationen der zwei dargestellten Klassen **`ElektronischesGeraet`** und **`Display`**. Fügen Sie nur für die Klasse **`Display`** den Konstruktor und Destruktor mit ein.

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 ElektronischesGeraet {
  protected:
    bool ist_eingeschaltet;

  public:
    void einschalten( );
};

class Display : public ElektronischesGeraet {
  private:
    float zoll;
    int pixelzahl;

  public:
    int getPixelzahl( );
    Display( );
    ~Display( );
};
```

Aufgabe 13: UML-Zustandsdiagramm

Im Folgenden wird ein Steuerungsablauf zum Einsähen von Pflanzensamen durch die Saatgutmaschine betrachtet (Bild 13.1).

Die Saatgutmaschine befindet sich nach Start im Zustand **`Abfragend`**. Bei Eintritt in diesen Zustand werden die Absperrschieber für die Behälter 1 und Behälter 2 aus Sicherheitsgründen einmalig geschlossen (**`iAbsperrer1 = 0`**, **`iAbsperrer2 = 0`**). Im Zustand **`Abfragend`** wartet die Saatgutmaschine auf eine Steuerungsanweisung (**`Abfragen()`**). Abhängig von der Steuerungsanweisung (**`command`**) wird bei der Steuerungsanweisung ('**`S`**') in den Zustand **`Säend`** und bei der Steuerungsanweisung ('**`M`**') in den Zustand **`Messend`** übergegangen.

Bei Eintritt in den Zustand **`Säend`** werden die Absperrschieber für die Behälter einmalig geöffnet (**`iAbsperrer1 = 1`**, **`iAbsperrer2 = 1`**). Das Saatgut fällt aufgrund der Schwerkraft in das vorgesehene Pflanzloch. Im Zustand **`Säend`** soll die Anzahl an eingesäten Pflanzensamen gezählt werden (**`Zählen()`**). Es sollen 5 Pflanzensamen als Sollvorgabe eingepflanzt werden. Die Anzahl der einzupflanzenden Pflanzensamen im Zustand **`Säend`** wird über eine zeitliche Steuerung festgelegt. Es kann aufgrund der Kalibrierung der Saatgutmaschine davon ausgegangen werden, dass im zeitgesteuerten Zustand **`Säend`** pro Sekunde 10 Pflanzensamen die Maschine verlassen. Für die Zeitsteuerung soll ein Timer (**`itimer`**) verwendet werden, welcher in Millisekunden-Schritten zählt (1000 Millisekunden entsprechen 1 Sekunde). Beim Verlassen des Zustandes **`Säend`** werden die Absperrschieber für die Behälter 1 und Behälter 2 aus Sicherheitsgründen wieder geschlossen (**`iAbsperrer1 = 0`**, **`iAbsperrer2 = 0`**).

Im Zustand **`Messend`** wird die Bodencharakteristik sensorisch erfasst (**`Messen()`**). Die Messung dauert 2 Sekunden.

Nach dem Einsäen oder Messen werden die Steuerungsdaten der Saatgutmaschine im Zustand **`Speichernd`** gespeichert (**`Speichern()`**). Das Speichern dauert 1 Sekunde lang, sodass sich die Saatgutmaschine anschließend wieder im Zustand **`Abfragend`** befindet.

Hinweis: Beim Verlassen aller Zustände soll die Timervariable **`itimer`** zurückgesetzt werden (**`itimer = 0`**).

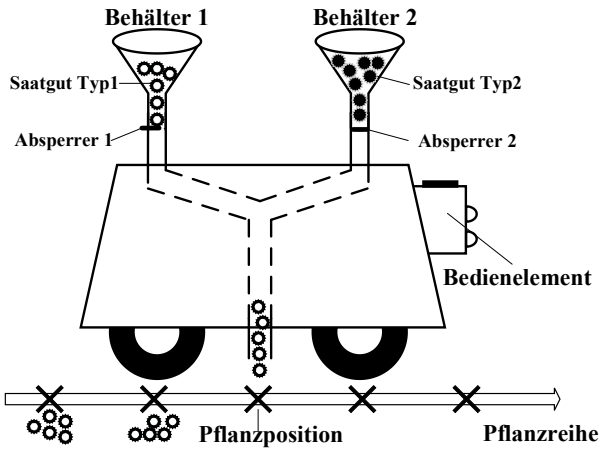


Bild 13.1: *Saatgutmaschine zum Einsähen von Pflanzensamen*

Es ist das in Bild 13.2 gezeigte Zustandsdiagramm mit den Zustandsnummern 1 bis 4 gegeben, welches den beschriebenen Pflanzvorgang abbildet.

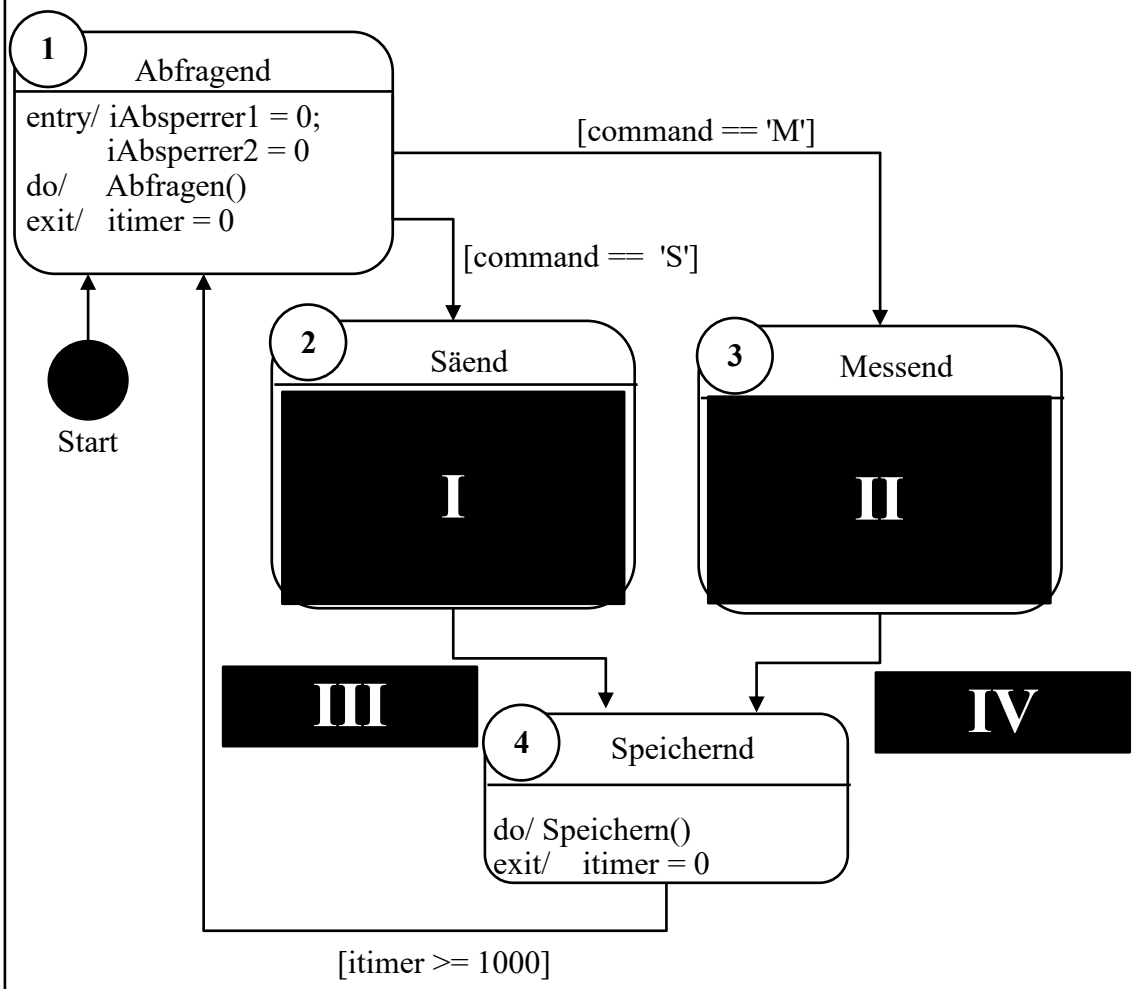


Bild 13.2: *Zustandsdiagramm*

Geben Sie untenstehend an, wie die durch römische Ziffern gekennzeichneten Lücken gefüllt werden müssen, um der Beschreibung in Aufgabe 13 zu entsprechen.

I. Modellieren Sie den Zustand 2 **Säend** korrekt aus:

```
entry/ iAbsperrer1 = 1;
       iAbsperrer2 = 1
do/    Zählen()
exit/  iAbsperrer1 = 0;
       iAbsperrer2 = 0;
       itimer = 0
```

II. Modellieren Sie den Zustand 3 **Messend** korrekt aus:

```
do/    Messen()
exit/  itimer = 0
```

III. Geben Sie die korrekte Wächterbedingung an:

[itimer >= 500] alternativ: [itimer == 500]

IV. Geben Sie die korrekte Wächterbedingung an:

[itimer >= 2000] alternativ: [itimer == 2000]

Aufgabe 14: UML-Zustandsdiagramm zu C-Code

Implementieren Sie Teile des in Aufgabe 13 modellierten Zustandsdiagramms in der Programmiersprache C. Nutzen Sie hierfür die in Tab. 14.1 vorgegebenen Variablen und vorimplementierten Funktionen.

Typ	Name	Beschreibung
VARIABLEN	`int istate`	Variable für den aktuellen Zustand der Saatgutmaschine mit insgesamt 4 Zuständen {1,2,3,4}.
	`int iAbsperrer1, int iAbsperrer2`	Variable zur Ansteuerung der Absperrer 1 und Absperrer 2 (`1`: offen; `0`: geschlossen).
	`unsigned int vplcZeit`	Zählvariable für die aktuelle Zeit der SPS in Millisekunden ab Programmstart.
	`unsigned int itime`	Zählvariable für Zeitmessung in Millisekunden.
FUNKTIONEN	`void Speichern()`	Funktion zum Speichern der Maschinendaten.

Tabelle 14.1: *Vorgegebene Variablen und vorimplementierte Funktionen*

Vervollständigen Sie das folgende Programmgerüst in der Programmiersprache C gemäß den Kommentaren im Lösungskästchen. Verwenden Sie hierfür die in Tabelle 14.1 angegebenen Variablennamen und Funktionen, welche im Header `Saatgutmaschine.h` bereits deklariert und implementiert sind.

Hinweis: Die Platzhalter /*ZUSTAENDE*/ enthalten spezifischen Code für Zustände des Zustandsautomaten und müssen nicht implementiert werden. Sie können für diese Aufgabe davon ausgehen, dass **`itimer`** beim Verlassen der Zustände `Säend` und `Messend` auf **`itimer = 0`** gesetzt wurde.

```
// C-Standard-Bibliothek stdio.h einbinden
    #include <stdio.h>
// Benutzerdefinierte Bibliothek Saatgutmaschine.h einbinden
    #include "Saatgutmaschine.h"
// Zustandsvariable istate
istate = 1;

int main () {
// Zyklische Endlosausfuehrung
    while(True) {
// Zustandsautomat
       switch(istate)
    {
     case 1: /* ZUSTAENDE */
     case 2: /* ZUSTAENDE */
     case 3: /* ZUSTAENDE */
     case 4:
                Speichern();
                if(itimer == 0) {
                   itimer = vplcZeit;
                }
                else {
                   if(vplcZeit-itimer >= 1000) {
                      itimer = 0;
                      istate = 1;
                   }
                }
                break;
     default: /* ZUSTAENDE */
    }
  }
return 0;
}
```

Aufgabe 15: Grundlagen in C

Gegeben ist nachfolgende Funktion **`funcA`**.

```
unsigned int funcA (unsigned int a)
{     int i = 42;
      for(int i = a; i > 0; i--) { printf("%d-", i);}
      return i;
}
```

a) Geben Sie den Rückgabewert der Funktion **`funcA`** bei Funktionsaufruf mit dem Übergabeparameter **`unsigned int a = 5`** an.

```
42
```

b) Geben Sie die Ausgabe der Funktion **`funcA`** bei Funktionsaufruf mit dem Übergabeparameter **`unsigned int a = 4`** an.

```
4-3-2-1-
```

c) Schreiben Sie in der Programmiersprache C eine Funktion **`funcB`**, die bei gleichem Übergabeparameter die gleiche Ausgabe erzeugt und den gleichen Wert rückgibt, wie die Funktion **`funcA`**, hierfür aber **keine** `for-Schleife` verwendet.

```
unsigned int funcB (unsigned int a)
{
   int i = 42;
   int j = a;
   while(j>0)
   {
        printf("%d-", j);
        --j; alt: j--; j=j-1;
   }
   return i;


}
```

Aufgabe 16: Algorithmen

An der Saatgutmaschine ist eine Wärmebildkamera zur Messung der Bodentemperatur montiert. Die Wärmebildkamera speichert die gemessene Temperaturverteilung des Bodens in einer Sensormatrix. Jedem Pixel wird ein Temperaturwert **als Gleitkommazahl** zugeordnet.

Für die Berechnung der Durchschnittstemperatur soll eine Funktion (**`mitteltemp`**) implementiert werden, welche die Durchschnittstemperatur für eine Sensormatrix mit der **Auflösung von 160x120 Pixeln** berechnet. Die Durchschnittstemperatur soll dabei über alle Pixel der Sensormatrix berechnet werden, mit Ausnahme der obersten und untersten Pixelzeile.

Die Funktion (**`mitteltemp`**) hat als Rückgabewert die **Durchschnittstemperatur als vorzeichenbehaftete Gleitkommazahl** und bekommt als Übergabeparameter übergeben:

- einen Zeiger (**`matrix`**), welcher auf das erste Element (an der Stelle [0][0]) einer zweidimensionalen Sensormatrix vom Typ (**`float`**) mit 160 Zeilen und 120 Spalten zeigt.

Implementieren Sie die Funktion (**`mitteltemp`**) in der Programmiersprache C, indem Sie das untenstehende Lösungskästchen ergänzen.

Hinweis: Sie müssen in der Funktion **`mitteltemp`** einen Zeiger verwenden, um auf die Elemente der Sensormatrix zuzugreifen.

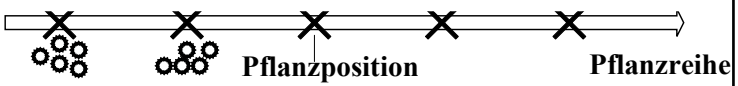
```
float mitteltemp( float* matrix )
{
 float temp = 0.0;

 for(int i = 1; i < 159; i++){
    for(int j = 0; j < 120; j++){

            temp += *(matrix + i*120 + j );
    }                    Alternativ: *(matrix++)
 }

 return temp/(160*120 - 2*120);
}
```

Aufgabe 17: Datenstrukturen



Für die Saatgutmaschine soll eine einfache Datenbankfunktionalität in Form einer **doppelt verketteten Liste mit Listenkopf** entworfen werden, um den Pflanzvorgang digital zu erfassen. Jede Pflanzposition wird genau einem Listenelement zugeordnet.

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

- Der Name des Beetes (**`name`**) als Zeichenkette mit exakt 10 Buchstaben. Berücksichtigen Sie dabei ein abschließendes Nullzeichen bei Zeichenketten.
- Ein Zeiger (**`pfirst`**) auf das erste Listenelement vom Typ **`PFLANZE`**.

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

- Anzahl der an einer Pflanzposition eingesäten Pflanzensamen (**`samen`**) als positive Ganzzahl. Berücksichtigen Sie, dass maximal 10 Samen eingepflanzt werden können. Achten Sie auf Speichereffizienz.
- Breitengrad (**breitengrad**) der Pflanzposition in Dezimalschreibweise als Gleitkommazahl einfacher Genauigkeit. Achten Sie auf Speichereffizienz.
- Längengrad der Pflanzposition in der Schreibweise *Grad*, *Minuten*, *Dezimalsekunden* in dieser Reihenfolge als eindimensionales Array (**laengengrad**). *Grad* und *Minuten* werden als Ganzzahl, *Dezimalsekunden* als Gleitkommazahl vom Typ **double** angegeben**.** Wählen Sie einen geeigneten Datentypen für das Array (**laengengrad**). Achten Sie auf Speichereffizienz.
- Ein Zeiger (**`pnext`**) auf das nachfolgende Listenelement.
- Ein Zeiger (**`pprev`**) auf das vorherige Listenelement.

a) Implementieren Sie ein Listenelement vom Typ **`PFLANZE`** sowie den Listenkopf vom Typ **`BEET`**, indem Sie die Lösungskästchen in der Programmiersprache C ausfüllen.

```
typedef struct pflanze {
    unsigned char samen;
    float breitengrad;
    double laengengrad[3];
    struct pflanze* pnext;
    struct pflanze* pprev;

} PFLANZE;
```

```
typedef struct{
    char name[11];          Alternativ: uint_8
    PFLANZE* pfirst;

} BEET;
```

Aufgrund von Störungen in der Saatgutmaschine kann es vorkommen, dass an einer Pflanzposition keine Pflanzensamen eingesät werden. In einer Funktion (**pruefen**) soll nach dem Saatvorgang für jedes Listenelement vom Typ **PFLANZE** in einer doppelt verketteten Liste überprüft werden, ob Pflanzensamen eingesät wurden (**samen > 0**). Sofern keine Pflanzensamen eingesät wurden, soll das Listenelement vom Typ **PFLANZE** aus der Liste entfernt werden.

Die Funktion **pruefen** hat keinen Rückgabewert und bekommt als Übergabeparameter einen Zeiger (**start**) auf den Listenkopf vom Typ **BEET** übergeben. Gehen Sie davon aus,

- dass **pnext** des letzten Listenelementes und **pprev** des ersten Listenelementes auf NULL zeigen
- dass **pfirst** des Listenkopfes auf **NULL** zeigt, sofern die Liste leer ist.
- dass immer mehr als 1 Listenelement existiert, sofern die Liste nicht leer ist.

Beim Löschen des ersten oder letzten Listenelements sollen **pnext** des dann letzten Listenelements bzw. **pprev** des dann ersten Listenelements weiterhin auf NULL zeigen.

b) Implementieren Sie die Funktion **pruefen** in der Programmiersprache C gemäß den Kommentaren im Lösungskästchen und der obigen Beschreibung.

```
void pruefen (BEET* start){
    PFLANZE* tmp = start->pfirst;
    while (tmp!=NULL) {
// zu loeschendes Element ist das 1. Element
        if(tmp->samen==0 && tmp->pprev==NULL){
            start->pfirst = tmp->pnext;
            tmp->pnext->pprev = NULL;
            free(tmp);
        }
// zu loeschendes Element ist das letzte Element
        else if(tmp->samen==0 && tmp->pnext==NULL){
            tmp->pprev->pnext = NULL;
            free(tmp);
        }
// zu loeschendes Element ist weder 1. noch letzte Element
        else if(tmp->samen==0){
            tmp->pprev->pnext = tmp->pnext;
            tmp->pnext->pprev = tmp->pprev;
            free(tmp);
        }
        tmp= tmp->pnext;
    }
}
```