Aufgabe 1: (12 Punkte)

Bei den Einfachauswahlfragen in dieser Aufgabe ist jeweils nur **eine** richtige Antwort eindeutig anzukreuzen. Auf die richtige Antwort gibt es die angegebene Punktzahl.

Wollen Sie eine Antwort korrigieren, streichen Sie bitte die falsche Antwort mit drei waagrechten Strichen durch (☒) und kreuzen die richtige an.

Lesen Sie die Frage genau, bevor Sie antworten.

a) Welche Aufgabe erfüllt der C-Präprozessor? 2 Punkte

- ☐ Er bindet mehrere .o-Dateien zusammen
- ☐ Er löst beim Binden die Referenzen zwischen Programm und Bibliotheken auf.
- ☐ Er führt textuelle Ersetzungen im C-Code durch, die sich durch Makrodefinitionen steuern lassen.
- ☐ Er entfernt vor dem Kompilieren ungenutzte Variablen aus dem Programm.

b) Worin liegt der Unterschied zwischen den wie folgt in der Datei `prog.c` deklarierten Funktionen? 2 Punkte

```
uint8_t testA(void);
static uint8_t testB(uint18_t param);
```

- ☐ Die Funktion `testA` ist nur für Funktionen in der Datei `prog.c` aufrufbar.
- ☐ Die Funktion `testB` wird vom Linker statisch in das Programm gebunden, `testA` wird hingegen dynamisch (zur Laufzeit) eingebunden.
- ☐ Die Funktion `testB` ist nur über einen Funktionszeiger aufrufbar.
- ☐ Die Funktion `testB` ist nur für Funktionen im selben Modul aufrufbar.

c) Gegeben sind folgende Makros: 2 Punkte

```
#define SUM(a,b) (a+b)
#define SQ(a) a*a
```

Was ist das Ergebnis des folgenden Ausdrucks?

```
SQ(SUM(3,5) + 2)
```

- ☐ 100
- ☐ 26
- ☐ 21
- ☐ 20

d) Welche der folgenden Aussagen bzgl. der Interruptsteuerung ist richtig? 2 Punkte

- ☐ Pegel-gesteuerte Interrupts werden beim Wechsel des Pegels ausgelöst.
- ☐ Flanken-gesteuerten Interrupts können nicht blockiert werden, da sie völlig unvorhersehbar auftreten.
- ☐ Während der Bearbeitung eines Interrupts nimmt der Prozessor keine weiteren Interrupts an.
- ☐ Wurde gerade ein Pegel-gesteuerter Interrupt ausgelöst, so muss erst ein Pegelwechsel der Interruptleitung stattfinden, bevor erneut ein Interrupt ausgelöst wird.

e) Welche Aussage zu Zeigern ist richtig? 2 Punkte

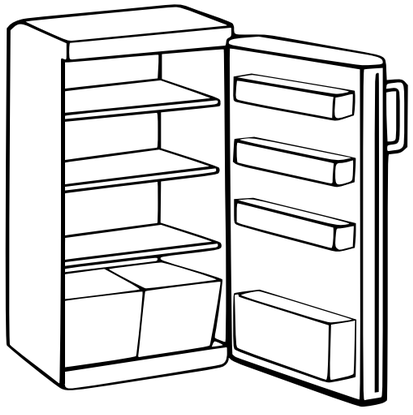
- ☐ Ein Zeiger kann zur Manipulation von schreibgeschützten Datenbereichen verwendet werden.
- ☐ Der Speicherbedarf eines Zeigers ist abhängig von der Größe des Objekts, auf das er zeigt.
- ☐ Zeiger vom Typ `(void *)` existieren in C nicht, da solche „Zeiger auf Nichts“ keinen sinnvollen Einsatzzweck hätten.
- ☐ Zeiger können verwendet werden, um in C eine call-by-reference Übergabesemantik nachzubilden

f) Welche Aussage zur Speicherallokation ist richtig? 2 Punkte

- ☐ Die Speicheradresse von statisch allokierten Variablen kann sich zur Laufzeit ändern.
- ☐ Die Verwendung von statisch allokierten Variablen erlaubt den Speicherbedarf bereits nach dem Binden abzuschätzen.
- ☐ Die dynamische Allokation von Speicher ist auf einem Mikrokontroller zu bevorzugen, da erst zur Laufzeit geprüft wird, ob der Speicher wirklich zur Verfügung steht.
- ☐ `automatic`-Variablen werden im Heap allokiert.

Aufgabe 2: Kühlschrank (30 Punkte)

Sie dürfen diese Seite zur besseren Übersicht bei der Programmierung heraustrennen!



Schreiben Sie ein Programm für den AVR-Mikrocontroller, welches die Tür und Temperatur bei einem Kühlschrank überwacht:

Sobald die Tür geöffnet wird, wird das Licht aktiviert. Wird die Tür geschlossen, wird das Licht wieder deaktiviert.

Außerdem wird in regelmäßigen Abständen (alle 30 Sekunden) die Temperatur gemessen und bei einer kontinuierlicher Überschreitung der Grenztemperatur (10 °C) von mindestens 5 Minuten (also 10 Messvorgängen) ein Warnsignal ausgegeben – bis einer der periodischen Messvorgänge eine Temperatur unter dem Grenzwert meldet.

Für das Warnsignal wird ein Piepser verwendet, der bei Aktivierung einen Ton erzeugt. Dabei soll das periodisch erklingende Warnsignal 1.5 Sekunden dauern, gefolgt von einer 1.5 Sekunden langen Pause.

Der Zustand der Tür – geschlossen oder offen – wird durch einen Schalter an der Tür erkannt: Sobald die Tür geöffnet wird, ist auch der Schalter geöffnet. Ist die Tür komplett geschlossen, schließt auch der Schalter.

Im Detail soll Ihr Programm wie folgt funktionieren:

- Initialisieren Sie die Hardware in der Funktion **`void`** `init(`**`void`**`)`. Treffen Sie hierbei keine Annahmen über den initialen Zustand der Hardware-Register.
- Die Abfrage der aktuellen Temperatur geschieht über die gegebene Bibliotheksfunktion **`int8_t`** `getTemp(`**`void`**`)`, welche die aktuelle Temperatur in Grad Celsius als vorzeichenbehaftete Ganzzahl zurück gibt. Während des Aufrufs müssen die Unterbrechungen freigegeben sein.
- Verwenden Sie die externe Interruptquelle 1 zur Erkennung von Türbewegungen.
- Für die Zeittaktung soll ein 8-Bit Timer so konfiguriert werden, dass er in der Lage ist $100ms$ Intervalle zu zählen: Konfigurieren Sie diesen so, dass er alle $T = 100ms = 0.1s$ einen Interrupt auslöst.
- Das Ansteuern des Piepsers soll in eine eigene Funktion **`void`** `warn(`**`uint8_t`** `on)` ausgelagert werden. Der Übergabeparameter `on` gibt an, ob gerade ein Warnbedingung aktiv ist (Temperatur zu hoch) oder nicht. Nutzen Sie zur Implementierung des periodischen Warnsignals keine globalen Variablen, sondern verwalten Sie nötigen Zustand innerhalb der Funktion `warn()` selbst.
- Stellen Sie sicher, dass sich der Mikrocontroller möglichst oft im Schlafmodus befindet um Strom zu sparen.

Information über die Hardware

Sie dürfen diese Seite zur besseren Übersicht bei der Programmierung heraustrennen!

Türschalter: Interruptleitung an **`PORTD`**, Pin 3

- Ist die Tür geschlossen liegt ein HIGH-Pegel an, ansonsten ein LOW-Pegel
- Pin als Eingang konfigurieren: Entsprechendes Bit im **`DDRD`**-Register auf 0
- Internen Pull-Up-Widerstand aktivieren: Entsprechendes Bit im **`PORTD`**-Register auf 1
- Externe Interruptquelle **`INT1`**, ISR-Vektor-Makro: **`INT1_vect`**
- Aktivieren/Deaktivieren der Interruptquelle erfolgt durch Setzen/Löschen des **`INT1`**-Bits im Register **`EIMSK`**
- Auslesen des Zustands über entsprechendes Bit im **`PIND`**-Registers

Konfiguration der externen Interruptquelle **`INT1`** (Bits im Register **`EICRA`**)

Interrupt 0		Beschreibung	Interrupt 1	
ISC01	ISC00		ISC11	ISC10
0	0	Interrupt bei low Pegel	0	0
0	1	Interrupt bei beliebiger Flanke	0	1
1	0	Interrupt bei fallender Flanke	1	0
1	1	Interrupt bei steigender Flanke	1	1

Licht: Ausgang an **`PORTB`**, Pin 4

- Bei anliegendem LOW-Pegel leuchtet das Licht
- Pin als Ausgang konfigurieren: Entsprechendes Bit im **`DDRB`**-Register auf 1
- Licht zunächst aus, entsprechendes Bit im **`PORTB`**-Register auf 1

Piepser: Ausgang an **`PORTC`**, Pin 6

- Bei anliegendem LOW-Pegel emittiert der Piepser den Warnton
- Pin als Ausgang konfigurieren: Entsprechendes Bit im **`DDRC`**-Register auf 1
- Piepser zunächst aus, entsprechendes Bit im **`PORTC`**-Register auf 1

Zeitgeber (8-bit): **`TIMER0`**

- Es soll die Überlaufunterbrechung verwendet werden (ISR-Vektor-Makro: **`TIMER0_OVF_vect`**)
- Der ressourcenschonendste Vorteiler (*prescaler*) ist 1024, wodurch es bei dem 2.6 MHz CPU-Takt alle $100ms$ zum Überlauf des 8-bit-Zählers **`TCNT0`** kommt.
- Aktivieren/Deaktivieren der Interruptquelle erfolgt durch Setzen/Löschen des **`TOIE0`**-Bits im Register **`TIMSK0`**

Konfiguration der Frequenz des Zeitgebers **`TIMER0`** (Bits im Register **`TCCR0B`**)

CS02	CS01	CS00	Beschreibung
0	0	0	Timer aus
0	0	1	CPU-Takt
0	1	0	CPU-Takt / 8
0	1	1	CPU-Takt / 64
1	0	0	CPU-Takt / 256
1	0	1	CPU-Takt / 1024
1	1	0	Ext. Takt (fallende Flanke)
1	1	1	Ext. Takt (steigende Flanke)

Ergänzen Sie das folgende Codegerüst so, dass ein vollständig übersetzbares Programm entsteht.

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include <stdint.h>

extern int8_t getTemp(void);    // Temperatur in Grad C

const uint8_t TEMP_LIMIT = 10;  // Grenztemperatur (10 Grad C)
const uint8_t TEMP_INTV  = 30;  // Temperatur Intervall (30s)

// Funktionsdeklarationen, globale Variablen, etc.

// Unterbrechungsbehandlungsfunktionen

// Ende Unterbrechungsbehandlungsfunktionen
```

U:


```
// Funktion main

  // Initialisierung und lokale Variablen

  // Hauptschleife

    // Warten auf Ereignisse

    // Ereignisse verarbeiten
```

```
// Ende Main
```

M:


```
// Warnfunktion
```

```
// Ende Warnfunktion
```

W:

```
// Initialisierungsfunktion
```

```
// Ende Initialisierungsfunktion
```

I:

Aufgabe 3: Bitoperationen (10 Punkte)

a) Kreuzen Sie an welche LEDs nach Aufruf von `func(0xcc)` leuchten.

2 Punkte

```
static void func(uint8_t leds) {
    sb_led_setMask(leds & 0x3c);
}
```

- ☐ RED0 (Bit 0)
- ☐ YELLOW0 (Bit 1)
- ☐ GREEN0 (Bit 2)
- ☐ BLUE0 (Bit 3)
- ☐ RED1 (Bit 4)
- ☐ YELLOW1 (Bit 5)
- ☐ GREEN1 (Bit 6)
- ☐ BLUE1 (Bit 7)

Notizen:

b) Kreuzen Sie an welche LEDs nach Aufruf von `func(4)` leuchten.

2 Punkte

```
static void func(uint8_t i) {
    if(i > 0) {
        sb_led_setMask((1 << i) | (1 << (i-1));
    }
}
```

- ☐ RED0 (Bit 0)
- ☐ YELLOW0 (Bit 1)
- ☐ GREEN0 (Bit 2)
- ☐ BLUE0 (Bit 3)
- ☐ RED1 (Bit 4)
- ☐ YELLOW1 (Bit 5)
- ☐ GREEN1 (Bit 6)
- ☐ BLUE1 (Bit 7)

Notizen:

c) Kreuzen Sie an welche LEDs nach Aufruf von `func(0xfe)` leuchten.

2 Punkte

```
static void func(uint8_t leds) {
    leds |= ~(1 << 7 | 1);
    sb_led_setMask(leds);
}
```

- ☐ `RED0` (Bit 0)
- ☐ `YELLOW0` (Bit 1)
- ☐ `GREEN0` (Bit 2)
- ☐ `BLUE0` (Bit 3)
- ☐ `RED1` (Bit 4)
- ☐ `YELLOW1` (Bit 5)
- ☐ `GREEN1` (Bit 6)
- ☐ `BLUE1` (Bit 7)

Notizen:

d) Beschreiben Sie welche LEDs nach Aufruf von `func()` leuchten. Sie müssen keine konkreten LEDs nennen. Beispielantworten: *die obersten vier LEDs*, *alle LEDs für die eine 1 in* `leds` *gesetzt ist*, *die unterste LED*.

2 Punkte

```
static void func(uint8_t leds) {
    sb_led_setMask(leds ^ 0xff);
}
```

e) Beschreiben Sie welche LEDs beim Aufruf `func()` je Iteration leuchten. Sie müssen keine konkreten LEDs nennen. Beispielantworten: *die obersten vier LEDs*, *alle LEDs für die eine 1 in* `leds` *gesetzt ist*, *die unterste LED*.

2 Punkte

```
static void func(void) {
    for(uint8_t i = 0; i < 4; i++) {
        sb_led_setMask(0xfe + i);
    }
}
```

`Iteration 1 (i=0):`

`Iteration 2 (i=1):`

`Iteration 3 (i=2):`

`Iteration 4 (i=3):`

Aufgabe 4: Nebenläufigkeit (8 Punkte)

Sie dürfen diese Seite zur besseren Übersicht heraustrennen!

Das Warenhaus ***Co**nvenience **Vi**tal **D**eluxe* (Covid) möchte auf Grund einer Pandemie die Zahl der gleichzeitig im Haus verweilenden Kunden begrenzen. Dazu hat es an allen Eingängen (Interrupt 0) als auch Ausgängen (Interrupt 1) Schranken aufgebaut, um die aktuell im Laden befindlichen Kunden zu zählen. Befinden sich 300 oder mehr Kunden im Laden, werden die Schranken an den Eingängen vorübergehend gesperrt, bis mindestens ein Kunde den Laden wieder verlassen hat.

Auf Grund manueller Zählungen ist bekannt, dass die Implementierung ein Nebenläufigkeitsproblem beinhalten muss.

```
1  #include <avr/interrupt.h>
2  #include <avr/sleep.h>
3
4  static volatile uint16_t visitors = 0;
5  static volatile uint8_t event = 0;
6  static void lock_entrance(uint16_t visitors, uint16_t threshold);
7
8  ISR(INT0_vect) { // Eingänge
9      visitors++;
10     event++;
11 }
12
13 ISR(INT1_vect) { // Ausgänge
14     visitors--;
15     event++;
16 }
17
18 void main(void) {
19     sleep_enable();
20     while(1) {
21         cli();
22         while(event == 0) {
23             sei();
24             sleep_cpu();
25             cli();
26         }
27         uint16_t l_visitors = visitors;
28         sei();
29
30         lock_entrance(l_visitors, 300);
31         event--;
32     }
33 }
```


a) Benennen Sie das Nebenläufigkeitsproblem und kennzeichnen Sie den kritischen Abschnitt im Quellcode. Skizzieren Sie einen konkreten Ablauf, der hier zu einem Nebenläufigkeitsfehler führt, und seine Auswirkungen (Stichpunkte, kurze Sätze). (7 Punkte)

---

b) Beschreiben Sie die notwendigen Änderungen am Programmcode (was und wo), um dieses Problem zu vermeiden. (1 Punkt)
