Aufgabe 1: (14 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 der folgenden Aussagen zum Schlüsselwort **`volatile`** ist richtig?

2 Punkte

- ☐ Das Schlüsselwort **`volatile`** erlaubt dem Compiler bessere Optimierungen durchzuführen.
- ☐ Das Schlüsselwort **`volatile`** beschleunigt den Zugriff auf die damit definierte Variable.
- ☐ Das Schlüsselwort **`volatile`** sorgt dafür, dass die damit definierte Variable nur so kurz wie möglich in einem Register gehalten wird.
- ☐ Das Schlüsselwort **`volatile`** unterbindet alle Nebenläufigkeitsprobleme in der entsprechenden Datei.

b) Welche der folgenden Aussagen über den C-Präprozessor ist richtig?

2 Punkte

- ☐ Der Präprozessor ist eine Softwarekomponente, welche Shell-Funktionen durch C-Funktionen ersetzt, die dann von einem C-Compiler übersetzt werden.
- ☐ Nach dem Übersetzen und dem Binden müssen C-Programme durch den Präprozessor nachbearbeitet werden, um Makros aufzulösen.
- ☐ Der Präprozessor optimiert Makros durch Zeigerarithmetik.
- ☐ Die Syntax von Präprozessoranweisungen ist unabhängig vom Rest der Sprache C.

c) Welche der folgenden Aussagen bzgl. der Interruptsteuerung ist richtig?

2 Punkte

- ☐ Pegel-gesteuerte Interrupts werden beim Wechsel des Pegels ausgelöst.
- ☐ Beim Auftreten eines Interrupts sichert der Prozessor automatisch Register.
- ☐ Wurde gerade ein Pegel-gesteuerter Interrupt ausgelöst, so muss erst ein Pegelwechsel der Interruptleitung stattfinden, bevor erneut ein Interrupt ausgelöst wird.
- ☐ Flanken-gesteuerten Interrupts können nicht blockiert werden, da sie völlig unvorhersehbar auftreten.

d) Welche der folgenden Aussagen zu Zeigern ist richtig?

2 Punkte

- ☐ Der Speicherbedarf eines Zeigers ist unabhängig von der Größe des Objekts, auf das er zeigt.
- ☐ Zeiger vom Typ `void*` benötigen weniger Speicher als andere Zeiger, da bei anderen Zeigertypen zusätzlich die Größe gespeichert werden muss.
- ☐ Zeiger vom Typ `void*` sind am besten für Zeigerarithmetik geeignet, da sie kompatibel zu jedem Zeigertyp sind.
- ☐ Ein Zeiger darf nie auf seine eigene Speicheradresse verweisen.

e) Gegeben ist folgender Programmcode:

2 Punkte

```
uint8_t foo[3] = { 23, 42 };
uint8_t *bar = &(foo[0]);
bar++;
```

Welchen Wert liefert die Dereferenzierung von `bar` (also `*bar`) nach der Ausführung des Programmcodes?

- ☐ 24
- ☐ 23
- ☐ Zur Laufzeit tritt ein Fehler auf.
- ☐ 42

f) Gegeben ist folgendes Makro:

2 Punkte

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

Wie ist das Ergebnis des folgenden Ausdrucks?

```
SUM(2, 3) * SUM(2 - 1, 4)
```

- ☐ 9
- ☐ 11
- ☐ 15
- ☐ 25

g) Welche der folgenden Aussagen zur Speicherorganisation auf einem Mikrocontroller ist richtig?

2 Punkte

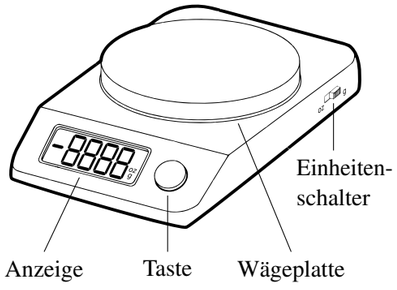
- ☐ Für jede lokale auto-Variable wird bereits beim Übersetzen exklusiv ein Speicherbereich reserviert.
- ☐ Eine dynamische Allokation ist auf einem Mikrocontroller nicht möglich.
- ☐ Im `.text`-Segment werden ausschließlich Zeichenketten gespeichert.
- ☐ Das `.bss`-Segment enthält ausschließlich mit 0 initialisierte Variablen.

Aufgabe 2: Digitalwaage (30 Punkte)

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

Eine batteriebetriebene digitale Haushaltswaage soll mit einem AVR-Mikrocontroller gesteuert werden.

Das aufliegende Gewicht soll über die vierstellige 7-Segmentanzeige ausgegeben werden. Das Gerät hat an der Vorderseite eine Taste: Diese wird sowohl zum Aktivieren (Aufwecken) der Waage aus dem Bereitschaftsmodus, als auch für die Tarier-Funktion (setzt das aktuelle Gewicht als Referenz/Null) verwendet.



Die Anzeige des aktuell aufliegenden Gewichts wird regelmäßig aktualisiert – bei längerer Inaktivität begibt sich die Waage in den stromsparenden Bereitschaftsmodus.

Auf der Seite ist ein Schalter zur Einheitenwahl angebracht.
Mit diesem kann von *Gramm* (`g`) auf *Unze* (`oz`) umgeschaltet werden.

Messprinzip: Durch das Gewicht verändert sich der elektrische Widerstand eines Messstreifens. Der elektrische Widerstand nimmt linear zur Gewichtskraft zu und wird mittels Analog/Digital-Wandler (ADC) gemessen.

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.
- Das Gewicht soll mittels ADC bestimmt werden:
  - Die ADC-Auflösung beträgt 14 Bit (d.h. Werte von 0 bis 16 383)
  - Für das Auslesen soll die Bibliotheksfunktion `uint16_t adc_read(void)` verwendet werden – diese darf **nicht im Interruptkontext** aufgerufen werden.
  - Bei zunehmenden Gewicht nimmt der Widerstand linear zu, dementsprechend fällt die Spannung und somit der Zahlenwert des ADCs linear ab.
  - Die Variablen `gramm_factor = 3` und `ounce_factor = 85` geben die Umrechnungsfaktoren an: Eine Veränderung des ADC-Wertes um `3` entspricht eine Gewichtsänderung von einem Gramm.
- Durch den Einheitenschalter soll die Anzeigeeinheit bestimmt werden.
- Die Bibliotheksfunktion `void display_number(int16_t value)` aktiviert die Anzeige und gibt einen Ganzzahl im Bereich von `-9999` bis `9999` aus.
- Ein Druck der Taste aktiviert die Waage und setzt das aktuelle Gewicht als Referenz: Der aktuelle Wert entspricht `0`, und alle nachfolgend gemessenen Werte werden relativ zu dieser Referenz ausgegeben (Tarier-Funktion).
- Unter Zuhilfenahme des 8-Bit Zeitgebers (*Timer*) soll die Anzeige alle **100 Millisekunden** aktualisiert werden.
- Nach einer Inaktivität von **10 Sekunden** ohne Gewichtsänderung oder Tastendruck sollen die Anzeige (mittels der Bibliotheksfunktion `void display_disable(void)`) und der Timer deaktiviert werden (Bereitschaftmodus). Ein Tastendruck aktiviert die Waage erneut.
- Um die Batterie der Waage zu schonen, soll auf Polling verzichtet werden und der Mikrocontroller so viel Zeit wie möglich im Schlafmodus verbringen.

Information über die Hardware

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

Einheitenschalter: **`PORTB`**, Pin 7

- In der Stellung *Gramm* liegt ein HIGH-Pegel an, bei *Unze* ein LOW-Pegel
- Pin als Eingang konfigurieren: entsprechendes Bit im **`DDRB`**-Register auf 0
- Externer Pull-Up-Widerstand: entsprechendes Bit im **`PORTB`**-Register auf 0
- Auslesen der Stellung über entsprechendes Bit im **`PINB`**-Register

Taste: Interruptleitung an **`PORTD`**, Pin 2

- active-low: Wird die Taste gedrückt, so liegt ein LOW-Pegel an
- 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 **`INT0`**, ISR-Vektor-Makro: **`INT0_vect`**
- Aktivierung der Interruptquelle erfolgt durch Setzen des **`INT0`**-Bits im Register **`EIMSK`**

Konfiguration der externen Interruptquelle **`INT0`** (Bits im Register **`EICRA`**)

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

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 16 MHz CPU-Takt alle $16.384\,\mathrm{ms}$ zum Überlauf des 8-bit-Zählers **`TCNT0`** kommt
- Somit entsprechen 6 Überlaufunterbrechungen etwa 100 Millisekunden (die Ungenauigkeit von $2.7\,\mathrm{ms}$ darf ignoriert werden)
- Aktivierung der Interruptquelle erfolgt durch Setzen 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.

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

extern uint16_t adc_read(void);
extern void display_disable(void);
extern void display_number(int16_t value);

const uint8_t gramm_factor = 3;
const uint8_t ounce_factor = 85;
```



```c
// Funktionsdeklarationen, globale Variablen, etc.
```

```c
// Unterbrechungsbehandlungsfunktionen
```

```c
// Ende Unterbrechungsbehandlungsfunktionen
```

A:

```
// Funktion main

  // Initialisierung und lokale Variablen

  // Hauptschleife

    // Warten auf Ereignisse
```

M:


```
    // Ereignisbehandlung
```

E:

```
// Ende main
// Initialisierungsfunktion
```

```
// Ende Initialisierungsfunktion
```

I:

Aufgabe 3: Programmiersprache C (9 Punkte)

Die folgenden Beschreibungen sollen kurz und prägnant erfolgen (Stichworte, kurze Sätze).

a) Was versteht man unter der Sichtbarkeit und Lebensdauer einer Variable und wie werden diese Konzepte in C umgesetzt? (6 Punkte)

b) Erklären Sie kurz die beiden Übergabesemantiken `Call-by-Value` und `Call-by-Reference` für Funktionsargumente. C nutzt standardmäßig die `Call-by-Value` Semantik. Wie kann die `Call-by-Reference` Semantik in C nachgebildet werden? (3 Punkte)

Aufgabe 4: Nebenläufigkeit (7 Punkte)

Sie sind im Bereich der Raketenwissenschaften tätig und sollen ein Überwachungssystem für eine sicherheitskritische Hardwarekomponente (den Lagesensor) prüfen. Dazu wird Ihnen nachfolgender Code für einen 8-Bit-AVR-Mikrocontroller vorgelegt, in dem der Sensor in regelmäßig Abständen über den externen Interrupt 0 die korrekte Funktionalität signalisiert. Sofern diese Mitteilung für eine bestimmte Zeit ausbleibt, ist von einem Fehlverhalten der Komponente auszugehen und das Sicherheitsprotokoll (Selbstzerstörung der Rakete) wird ausgelöst.

Aufgrund der Auswertung der Protokolldaten von teuren Fehlstarts ist bekannt, dass die Implementierung ein Nebenläufigkeitsproblem beinhalten muss.

```
1  #include <avr/interrupt.h>
2
3  #define THRESHOLD 0x42
4  static volatile uint8_t value = 0;
5
6  ISR(INT0_vect) {
7      value = 0;
8  }
9
10 void main(void) {
11     /* ... */
12     sei();
13     while (1){
14         uint8_t local_value = value++;
15         if (local_value > THRESHOLD) {
16             // destroy rocket ...
17         }
18         // do something else ...
19     }
20 }
```

a) Benennen Sie das Nebenläufigkeitsproblem und kennzeichnen Sie den kritischen Abschnitt im Quellcode. (2 Punkte)

b) Skizzieren Sie einen konkreten Ablauf, der hier zu einem Nebenläufigkeitsfehler führt, und seine Auswirkungen (Stichpunkte, kurze Sätze). (4 Punkte)

c) Beschreiben Sie die notwendigen Änderungen am Programmcode, um dieses Problem zu vermeiden. (1 Punkt)