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 Eigenschaften muss der Vorteiler (*Prescaler*) eines Zeitgebers (*Timer*) für periodische Interrupts haben, um möglichst energieeffizient zu sein.

2 Punkte

- ☐ Der energieeffizienteste Vorteiler hängt von der Zählrichtung des Zeitgebers ab: möglichst klein für herunterzählende und möglichst groß für hochzählende Zeitgeber.
- ☐ Der Vorteiler hat keinen Einfluss auf die Energieeffizienz des Mikrocontrollers.
- ☐ Es sollte der kleinstmögliche Vorteiler gewählt werden, der ausreichend für die geforderte Genauigkeit ist.
- ☐ Es sollte der größtmögliche Vorteiler gewählt werden, der ausreichend für die geforderte Genauigkeit ist.

b) Was versteht man beim Zugriff auf I/O-Register unter dem Begriff "Memory-mapped"?

2 Punkte

- ☐ Die Register sind nicht real, sondern nur virtuell im Hauptspeicher vorhanden (sog. "virtual devices").
- ☐ Der Zugriff auf die Register erfolgt mit speziellen mmap-Instruktionen des Prozessors.
- ☐ Beim Zugriff auf spezielle Speicherbereiche des Hauptspeichers werden die Inhalte der Hauptspeicherzellen automatisch in Geräteregister umkopiert.
- ☐ Die Register sind in den normalen Adressraum des Prozessors eingeblendet und der Zugriff erfolgt mit den normalen Speicherzugriffsinstruktionen.

c) Welche der folgenden Aussagen zum Begriff der Rücksprungadresse ist richtig?

2 Punkte

- ☐ Bei rekursiven Funktionsaufrufen erstellt der Compiler eine Rücksprungadresse um sicher zu stellen, dass die Rekursion terminiert.
- ☐ Bei Aufruf einer Funktion über einen Funktionszeiger muss der Programmierer eine Rücksprungadresse angeben, an der das Programm später fortgesetzt werden soll.
- ☐ Die Rücksprungadresse ermöglicht die Rückkehr ins Betriebssystem. Auf einer Mikrocontroller-Plattform ist sie allerdings nicht vorhanden.
- ☐ Bei Aufruf einer Funktion sichert der Prozessor selbsttätig die Adresse der folgenden Instruktion. Dies ist die Rücksprungadresse.

d) Gegeben sei folgendes Programmfragment für einen AVR-Mikrocontroller:

2 Punkte

```
uint8_t a = 100;
uint8_t b;
b = a+a * 2-50;
```

Welche der folgenden Aussagen ist richtig?

- ☐ Der Compiler warnt zur Übersetzungszeit vor einem Bereichsüberlauf.
- ☐ `b` hat nach Ausführung der Zuweisung den Wert `250`.
- ☐ `b` hat nach Ausführung der Zuweisung den Wert `350`.
- ☐ Während der Ausführung kommt es zu einem Bereichsüberlauf; auf der AVR-Plattform bleibt dieser jedoch unentdeckt.

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

2 Punkte

- ☐ Interrupts sind eine Besonderheit von AVR-Mikroprozessoren. Auf anderen Architekturen müssen externe Ereignisse durch Polling abgefragt werden.
- ☐ Wurde gerade ein flankengesteuerter Interrupt ausgelöst, so muss erst ein Pegelwechsel der Interruptleitung stattfinden, damit erneut ein Interrupt ausgelöst werden kann.
- ☐ Pegelgesteuerte Interrupts müssen durch Polling des Pegels abgefragt werden.
- ☐ Pegelgesteuerte Interrupts werden bei jedem Wechsel des Pegels ausgelöst.

f) Welche Aussage zur Speicherallokation ist richtig?

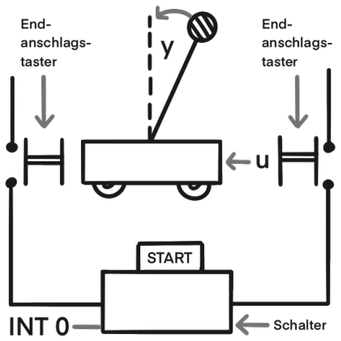
2 Punkte

- ☐ 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.
- ☐ `automatic`-Variablen werden im Heap allokiert.
- ☐ Die dynamische Allokation von Speicher ist auf einem Mikrokontroller zu bevorzugen, da erst zur Laufzeit geprüft wird, ob der Speicher wirklich zur Verfügung steht.

Aufgabe 2: Inverses Pendel (30 Punkte)

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

Implementieren Sie die Regelung für ein inverses Pendel. Dabei handelt es sich um einen klassischen Laborversuch, bei dem ein Stab drehbar auf einem Schlitten gelagert ist. Das Ziel ist es, den Stab durch Bewegung des Schlittens aufrecht zu halten. Dafür kann der Winkel des Stabes gemessen und eine Kraft auf den Schlitten ausgeübt werden. Ihr Regler soll, wie weiter unten beschrieben, periodisch die verrauschte Messung des Winkels filtern und eine dazu proportionale Kraft auf den Schlitten ausüben. Berührt der Schlitten einen der beiden Endanschlagstaster, soll keine Kraft mehr ausgeübt werden, bis mittels des Startknopfs ein neues Experiment gestartet wurde.



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.
- Der Eingang `PD2` (Interrupt 0) ist mit den Endanschlagstastern und dem Startknopf verbunden. Die externe Beschaltung stellt sicher, dass genau dann eine fallend Flanke auftritt, wenn der Schlitten einen Endanschlag berührt und eine steigende Flanke dann, wenn der Startknopf in einem gültigen Zustand gedrückt wird.
- Für die Zeittaktung soll ein 8-Bit Timer verwendet werden. Konfigurieren Sie diesen so, dass er alle $T = 1ms$ einen Interrupt auslöst.
- Der Regler soll alle T Zeiteinheiten ausgeführt werden. Messen Sie dafür zunächst durch Aufruf von **`int16_t`** `sb_adc_read(ADCDEV)` für `POTI` den Winkel y des Stabes als vorzeichenlose 10 Bit Ganzzahl. Während des Aufrufs müssen die Interrupts gesperrt sein.
- Implementieren Sie das Regelgesetz in der Funktion **`int16_t`** `compute(`**`int16_t`**`)`, welche den ausgelesenen Winkel entgegennimmt, ihn zunächst filtert, dann die zu stellende Kraft berechnet und diese zurückgibt. Der Tiefpassfilter soll dabei entsprechend der Gleichung

  $$f \leftarrow \frac{905}{1000} \cdot f + \frac{95}{1000} \cdot (y - 512)$$

  aktualisiert werden, wobei f der Filterzustand und y der im aktuellen Zeitschritt gemessene Winkel ist. Initialisieren Sie f mit 0. Der Messversatz zum Nullwinkel ist in der obenstehenden Gleichung bereits berücksichtigt. Die Sollkraft u berechnet sich dann proportional zum aktuellen Filterzustand:

  $$u \leftarrow -2 \cdot f.$$

- Die vom Regler berechnete Sollkraft u darf nur während eines laufenden Experiments (nach Drücken des Startknopfs und vor Berührung eines Anschlagtasters) als Kraft auf den Schlitten aufgeschalten werden. Übergeben Sie dafür den Betrag von u an die Funktion **`void`** `actuate(`**`uint16_t`**`)`. Ferner muss `PB0` bei negativem Vorzeichen von u gesetzt und anderenfalls gelöscht werden. Nach Erreichen des Endanschlags darf keine Kraft mehr ausgeübt werden. Rufen Sie stattdessen periodisch `actuate(0)` auf.
- Führen Sie alle Berechnungen ganzzahlig durch und achten Sie auf die korrekten Breiten der verwendeten Datentypen.
- Stellen Sie sicher, dass sich der Mikrocontroller möglichst oft im Schlafmodus befindet.

Information über die Hardware

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

Endanschläge und Startknopf: Interruptleitung an **`PORTD`**, Pin 2

- Steigende Flanke: Startknopf wird in einem gültigen Zustand gedrückt, Experiment beginnt
- Fallende Flanke: Schlitten berührt einen der beiden Endanschlagstaster, Experiment endet
- Pin als Eingang konfigurieren: Entsprechendes Bit im **`DDRD`**-Register auf 0
- Internen Pull-Up-Widerstand deaktivieren: Entsprechendes Bit im **`PORTD`**-Register auf 0
- Externe Interruptquelle **`INT0`**, ISR-Vektor-Makro: **`INT0_vect`**
- Aktivieren/Deaktivieren der Interruptquelle erfolgt durch Setzen/Löschen 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

Richtungssignal für ausgeübte Kraft: Ausgang an **`PORTB`**, Pin 0

- Gibt Richtung der ausgeübten Kraft vor. Für negative Kraft auf HIGH setzen, sonst auf LOW
- Pin als Ausgang konfigurieren: Entsprechendes Bit im **`DDRB`**-Register auf 1
- Ausgeübte Kraft zunächst 0, entsprechendes Bit im **`PORTB`**-Register auf 0

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

- Es soll die Überlaufunterbrechung verwendet werden (ISR-Vektor-Makro: **`TIMER0_OVF_vect`**)
- Der ressourcenschonendste Vorteiler (*prescaler*) ist 64, wodurch es bei dem 16 MHz CPU-Takt (hinreichend genau) alle $1ms$ 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/interrupt.h>
#include <avr/io.h>
#include <avr/sleep.h>
#include <stdint.h>
#include <adc.h>

extern int16_t sb_adc_read(ADCDEV dev);
extern int16_t actuate(uint16_t force);

// Parameter fuer die Filter- und Reglergleichungen
static const int16_t OFFSET = 512; // Korrekturwert fuer Nullwinkel
static const int16_t NUM_F = 905;  // Zaehler fuer Filterkoeffizient von f
static const int16_t NUM_Y = 95;   // Zaehler fuer Filterkoeffizient von y
static const int16_t DEN = 1000;   // Nenner fuer beide Filterkoeffizienten
static const int16_t K = 2;        // Proportionale Reglerverstaerkung

// Funktionsdeklarationen, globale Variablen, etc.

// Unterbrechungsbehandlungsfunktionen

// Ende Unterbrechungsbehandlungsfunktionen
```

D:


```
// Funktion main

  // Initialisierung und lokale Variablen

  // Hauptschleife

    // Ereignisse verarbeiten
```

```
// Ende main
```

M:


```
// Regelgesetz
```

```
// Ende Regelgesetz
```

R:

```
// Initialisierungsfunktion
```

```
// Ende Initialisierungsfunktion
```

I:

Aufgabe 3: Systemprogrammiersprache C (10 Punkte)

a) Die nativen Typen in der Programmiersprache C (`int, long, ...`) haben keine definierte Breite. Begründen Sie, wieso wir für die Mikrocontrollerprogrammierung auf die in `stdint.h` definierten Typen fester Breite (bspw. `uint16_t`) zurückgreifen. (2 Punkte)

b) Betrachten Sie folgenden Codeausschnitt. Nennen Sie alle möglichen Werte, die `local` nach Ausführen der Zeile 8 unter folgenden Annahmen annehmen kann: 1) `val` hat zum Funktionseintritt in `update` (vor Zeile 8) den Wert 255. 2) Interrupt **`INT0`** kann zu jedem Zeitpunkt, aber insgesamt **maximal einmal** auftreten. 3) Das Programm läuft ohne Compileroptimierungen auf dem aus Vorlesung und Übung bekannten 8-Bit AVR Mikrocontroller. (2 Punkte)

```
1  static volatile uint16_t val;
2
3  ISR(INT0_vect) {
4    val += 1;
5  }
6
7  void update (void) {
8    uint16_t local = val;
9    while (local > 42 ) {
10     // ...
11     local = val;
12   }
13 }
```

c) Welche Funktion erfüllt das Schlüsselwort **`volatile`** in obigem Beispielcode? Warum ist es hier notwendig? (1 Punkt)

Sie dürfen diese Seite zur besseren Übersicht heraustrennen!

print.h:

```
1  void print_msg(char *msg);
```

exam.h:

```
1  #include "print.h"
2  void init_arr(uint8_t pins, uint8_t len);
```

exam.c:

```
1  #include "exam.h"
2
3  #define LEVEL_INFO  1
4  #define LEVEL_DEBUG 2
5
6  #define LEVEL LEVEL_INFO
7
8  #if LEVEL >= LEVEL_INFO
9  #define INFO(msg) printf(msg)
10 #else
11 #define INFO(msg)
12 #endif
13
14 #if LEVEL >= LEVEL_DEBUG
15 #define DEBUG(msg) printf(msg)
16 #else
17 #define DEBUG(msg)
18 #endif
19
20 #define REPEAT(i,n) for(uint8_t i = 0; i < n; i++);
21
22 void init_arr(uint8_t pins, uint8_t len) {
23   INFO("init_arr");
24   REPEAT(i,len) {
25     DEBUG("init");
26     init(pin[i]);
27   }
28 }
```

d) Lösen Sie auf der nachfolgenden Seite die C-Präprozessordirektiven für den C-Code der obenstehenden Datei **exam.c** vollständig auf. (5 Punkte)

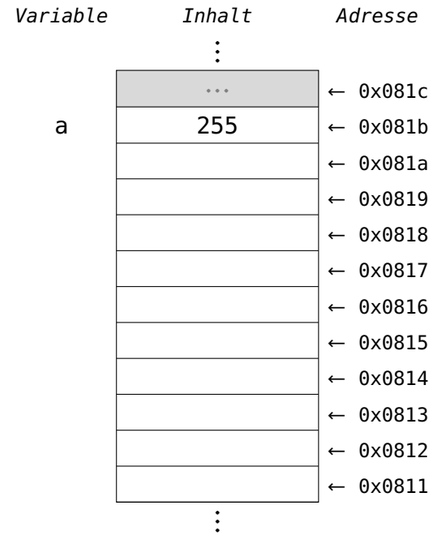
Aufgabe 4: Speicherorganisation (8 Punkte)

Das folgende Programm wird ohne Optimierungen übersetzt und auf einem 8-Bit AVR Mikrocontroller ausgeführt.

Hinweis: Lesen Sie zuerst die Aufgabenstellung – ein vollständiges Verständnis des Programms ist zur Bearbeitung der Aufgabe nicht notwendig.

```
1  #include <stdint.h>
2
3  static int8_t *max(const uint8_t array[], uint8_t size) {
4    int8_t m = -1;
5    for(uint8_t i = 0; i < size; i++) {
6      if(array[i] > m) {
7        m = array[i];
8      }
9    }
10   return &m;
11 }
12
13 void main(void) {
14   uint8_t a = 255;
15
16   // Aufgabe a)
17   uint8_t b[3] = {0, 8, 15};
18   uint8_t *c = &a;
19   uint16_t d = (uint16_t)(b[a%3]) + *c;
20   uint8_t e = a >> 4;
21   // Ende Aufgabe a)
22
23   // Aufgabe b)
24   uint8_t l[8] = {1,2,3,4,5,6,7,8};
25   int8_t *m1 = max(b, 3);
26   int8_t *m2 = max(l, 8);
27   if (*m1 > *m2) {
28     /* ... */
29   }
30
31   /* ... */
32 }
```

Dazugehöriger Stack (Auszug):



a) Obige Grafik zeigt einen Speicherauszug des Stacks nach der Ausführung der Zuweisung in Zeile `14` an. Erweitern Sie die Grafik um einen möglichen Aufbau des Stacks (Variable und Inhalt) nach der Ausführung der Zeilen `17` bis `20`. (4 Punkte)

b) Die Funktion `max()` aus obigem Codebeispiel soll einen Zeiger auf das größte Element eines Feldes (Arrays) zurückliefern. Welches Problem kann bei der Verarbeitung der Rückgabewerte von `max()` (Zeile `27`) auftreten? Wieso? (2 Punkte)

c) Wieso wird ein Programm, das eine Funktion in Endlosrekursion selbst aufruft, üblicherweise relativ schnell zu einem Systemabsturz oder ähnlichem Fehlverhalten führen anstatt tatsächlich endlos zu laufen? (2 Punkte)