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) Gegeben sei folgender Programmcode. Was gibt das Programm aus? 2 Punkte

```
char s1[] = "SPiC";
char s2[] = "SPIC";
s2[2] = 'i';

if(s1 == s2) {
    printf("match");
} else {
    printf("no match");
}
```

- ☐ Der C-Compiler meldet beim Übersetzen einen Fehler.
- ☐ Das Programm gibt `no match` aus.
- ☐ Das Programm stürzt zur Laufzeit ab.
- ☐ Das Programm gibt `match` aus.

b) Gegeben ist folgender Programmcode: 2 Punkte

```
int32_t x[] = {3, 8, -13, 5, 4};
int32_t *y = &x[4];
y -= 3;
```

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

- ☐ 8
- ☐ 1
- ☐ Zur Laufzeit tritt ein Fehler auf.
- ☐ 4

c) Gegeben ist folgender Ausdruck: 2 Punkte

```
if ( (a = 5) || (b != 3) ) ...
```

Welche Aussage ist richtig?

- ☐ Der initiale Wert von `a` hat keinen Einfluss auf das Ergebnis.
- ☐ Falls `a` den Wert `5` und `b` den Wert `7` enthält, wird falsch zurückgeliefert.
- ☐ Der Compiler meldet einen Fehler, weil dieser Ausdruck nicht zulässig ist.
- ☐ Falls `a` den Wert `7` und `b` den Wert `5` enthält, wird falsch zurückgeliefert.

d) Wie löst man das Nebenläufigkeitsproblem „Lost-Update“ zwischen Hauptprogramm und Interrupthandler auf einem Mikrocontroller?

2 Punkte

- ☐ Die Verwendung des Schlüsselwortes `volatile` löst alle Nebenläufigkeitsprobleme.
- ☐ Durch den Aufruf einer Callback-Funktion im Interrupthandler.
- ☐ Durch Synchronisation mittels kurzzeitigem Sperren der Interrupts.
- ☐ Durch die Verwendung von pegelgesteuerten anstelle von flankengesteuerten Interrupts.

e) 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?

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

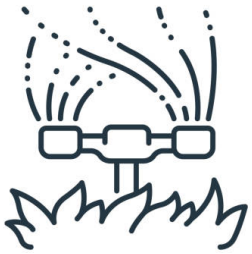
f) Was versteht man unter Polling?

2 Punkte

- ☐ Wenn ein Programm regelmäßig eine Peripherie-Schnittstelle abfragt, ob Daten oder Zustandsänderungen vorliegen.
- ☐ Wenn ein Gerät so lange Interrupts auslöst, bis die Daten durch den Mikrocontroller abgeholt wurden.
- ☐ Wenn ein Gerät durch Auslösen eines Interrupts Daten von einem Mikrocontroller anfordert.
- ☐ Wenn ein Programm zum Zugriff auf kritische Daten Interrupts sperrt.

Aufgabe 2: Bewässerungsanlage (30 Punkte)

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



Implementieren Sie die Steuerung einer automatischen Bewässerungsanlage, die automatisch Pflanzen gießen soll. Die Bedienung soll möglichst einfach sein: der Knopf weckt die Anlage aus dem Tiefschlaf, der Lichtsensor misst die Helligkeit und errechnet anhand der Helligkeit eine Gießdauer. Wenn die Gießdauer berechnet wurde, wird sie auf der Siebensegmentanzeige angezeigt und das Wasser für die errechnete Zeit in Minuten eingeschaltet. Während des Wässerns soll die blaue Kontrollleuchte (`BLUE0`) leuchten und die Siebensegmentanzeige die verbleibende Zeit anzeigen. Nach verstreichen der vorher errechneten Zeit, soll sich die Anlage von selbst wieder in den Schlafzustand begeben und auf die nächste Eingabe warten.

Das Programm soll im Detail 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 dem Taster verbunden. Eine fallende Flanke tritt genau dann auf, wenn der Taster gedrückt wird und eine steigende Flanke dann, wenn dieser wieder losgelassen wird. Sie dürfen davon ausgehen, dass der Taster initial nicht gedrückt gehalten wird.
- Für die Zeittaktung soll ein 8-Bit Timer verwendet werden. Konfigurieren Sie den sparsamsten Prescaler und lösen Sie einmal pro Sekunde ein Event aus. Auf der nächsten Seite finden Sie Details hierzu.
- Wenn der Taster gedrückt wurde, soll die Helligkeit mittels eines Aufrufs von `int16_t sb_adc_read(ADCDEV dev)` für das ADC-Gerät `PHOTO` bestimmt werden. Diese Funktion liefert eine vorzeichenlose 10-Bit-Ganzzahl als Rückgabewert. Während des Aufrufs müssen die Interrupts gesperrt sein. Berechnen Sie daraus einen Wert zwischen 3 Minuten für die minimale Helligkeit und 14 Minuten bei maximaler Helligkeit in Sekunden und geben Sie diesen zurück. Als Hilfestellung folgt ein Hinweis.
- **Hinweis**: Der Wertebereich der gemessenen Helligkeit ist `0` bis `1023`. So könnten Sie den gemessenen Wert ganzzahlig durch `100` teilen, um die Helligkeit auf ein Intervall $[0, 11]$ abzubilden. Auf diesen normierten Wert können Sie dann geeignet einen Offset addieren, um einen Wert auf das Zielintervall abzubilden.
- Zeigen Sie nun die errechnete Dauer in Minuten auf der 7-Segment Anzeige an.
- Setzen Sie den Ausgang `PD7` damit die blaue Status-LED `BLUE0` leuchtet, um den laufenden Betrieb zu signalisieren.
- Nutzen Sie die vorgegebenen Funktionen **`void`** `watering_on(`**`void`**`)` und **`void`** `watering_off(`**`void`**`)` aus der Datei `watering.h` um die Bewässerung an- und auszuschalten.
- Nutzen Sie die Funktion **`uint8_t`** `sb_7seg_showNumber(`**`uint8_t`**`)`, um die Gießdauer auf der Siebenzegmentanzeige anzuzeigen und **`void`** `sb_7seg_disable(`**`void`**`)`, um die Siebensegmentanzeige nach dem Gießen wieder auszuschalten.
- Nach Ablauf der Bewässerungszeit sollen Wasser, Siebensegmentanzeige und Kontrollleuchte ausgeschaltet und der Mikrocontroller wieder schlafen gelegt werden, bis der Taster erneut gedrückt wird.

Information über die Hardware

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

Taster: Interruptleitung an **PORTD**, Pin 2

- Fallende Flanke: Taster wird gedrückt
- Steigende Flanke: Taster wird losgelassen
- 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**
- 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

Betriebs-LED: Ausgang an **PORTD**, Pin 7

- Zeigt, dass die Bewässerung gerade läuft.
- Pin als Ausgang konfigurieren: Entsprechendes Bit im **DDRD**-Register auf 1
- Kontrollleuchte initial ausschalten, entsprechendes Bit im **PORTD**-Register auf 1

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/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include <stdint.h>
#include <adc.h>
#include "watering.h"

static const uint8_t OVFS_PER_SECOND = 61;

// Function Declarations, Global Variables, etc.
```

```
// End Function Declarations, Global Variables, etc.
// Interrupt Service Routines
```

```
// End Interrupt Service Routines
```

D:

```
// Function main

  // Initialization and Local Varibables

  // Event Loop
```

```
    // Process Button Event
```

```
    // Process Timer Event
```

```
// End main
```

M:

```
// Initialization Function
```

```
// End Initialization Function
```

I:

Aufgabe 3: lazy (21 Punkte)

Wiederkehrende Aufgaben, wie zum Beispiel die Veröffentlichung der Aufgaben, automatische Plagiatschecks oder Korrekturen, gehören zum festen Bestandteil des Übungsbetrieb. Um den Übungsbetrieb weiter zu automatisieren, implementieren Sie ein Programm `lazy`, das einen gegebenen Befehl wiederholt ausführt. Die ungefähre Wartedauer zwischen den Aufrufen soll ebenfalls als Kommandozeilenargument übergeben werden.

```
# # Führe den Befehl "plagiatscheck.sh blink" alle 300 Sekunden aus
$> ./lazy 300 plagiatscheck.sh blink
```

Das Programm soll im Detail wie folgt funktionieren:

- Das Programm prüft zu Beginn, ob mindestens zwei Parameter übergeben wurden. Sollte dies nicht der Fall sein, gibt es eine entsprechende Fehlermeldung aus und beendet sich.
- Wurde das Programm korrekt aufgerufen, soll mittels `sigaction` die benötigten Behandlungsroutinen für die Signale `SIGALRM` und `SIGCHLD` registriert werden. Diese sollen jeweils eine entsprechende Event-Variable setzen.
- Mit Hilfe der externen Hilfsfunktionen `parse_pos_integer`, soll die übergebene Wartedauer geparst werden.
- Anschließend soll die geparste Wartedauer an die ebenfalls externe Funktion `start_timer` übergeben werden. Diese starten einen periodischen Timer, der nach Ablauf des Intervalls dem aufrufenden Prozess ein `SIGALRM` zustellt.
- Warten Sie nun passiv mittels `sigsuspend` bis mindestens eine der beiden Event-Variablen gesetzt wurde.
- Falls zwischenzeitlich ein `SIGALRM` zugestellt wurde, erzeugen Sie einen neuen Kindprozess (`fork`) und führen Sie das übergeben Programm mit dessen Argumenten (`exec`) aus. Im Fehlerfalls soll nur eine entsprechende Warnung ausgegeben werden, und *nicht* beendet werden.
- Sind in der Zwischenzeit Kindprozesse terminiert und dies entsprechend signalisiert worden, sollen deren verwendete Prozessressourcen in einer Schleife mittels `waitpid` freigegeben werden.

Achten Sie auf eine korrekte Fehlerbehandlung der verwendeten Funktionen. Fehlermeldungen sollen generell auf `stderr` erfolgen.

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

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <unistd.h>

// Converts a string representation into the corresponding number
// expects numbers in base 10, no other base is supported.
// All invalid inputs are handled internally.
unsigned long parse_pos_integer(const char *str);

// Start a periodic timer for a given interval.
void start_timer(unsigned long seconds);

// Prints a show usage overview.
static void usage(const char *program) {
  fprintf(stderr, "Usage: %s SECONDS CMD [ARGS...]\n", program);
}
```

// Global Variables, Signal Handlers

D:

```
// Function main

  // Check Arguments

  // Prepare SIGALRM/SIGCHLD Handlers

  // Parse SECONDS Argument

  // Start Periodic Alarms
```

```
  // Signal Mask for Synchronisation
```

```
  // Endless while Loop
```

```
    // Passive Waiting Loop
```

```
    // Handle SIGALRM
```

```
    // Handle SIGCHLD
```

```
// End main
```

M:

Aufgabe 4: Systemprogrammiersprache C (9 Punkte)

a) Betrachten Sie folgenden Codeausschnitt. Beantworten Sie in Stichpunkten die folgenden Fragen:

1. Welche Bedeutung hat das Schlüsselwort **`static`** in Zeile 1?
2. Welche Bedeutung hat **`static`** in Zeile 2?
3. Welche Bedeutung hat das Schlüsselwort **`volatile`** in Beispielcode?
4. Begründen Sie, wieso wir für die Mikrocontrollerprogrammierung auf die in `stdint.h` definierten Typen fester Breite (bspw. `uint16_t`) zurückgreifen.

(4 Punkte)

```
1  static void init(void) {
2      static uint8_t initDone = 0;
3      if (initDone == 0) {
4        initDone = 1;
5        ...
6      }
7  }
8
9  void mod_func(void) {
10     init();
11     for(uint8_t i = 0; i < 5; i++) {
12         for(volatile uint16_t j = 0; j < 65000; j++);
13         sb_led_toggle(YELLOW0);
14     }
15     ...
16 }
```

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

print.h:

```
1  #ifdef NO_PRINT
2  #define PRINT(msg)
3  #else
4  #define PRINT(msg) printf(msg)
5  #endif
6
7  void printf(const char* msg);
```

exam.c:

```
1  #define WAIT
2  #define __AVR__
3  #define NO_PRINT
4
5  #include "print.h"
6
7  #ifdef __AVR__
8  void main(void) {
9      sei();
10     pointerDemo();
11 #else
12 int main(void) {
13     return pointerDemo();
14 #endif
15 }
16
17 static void wait_msg(const char *msg) {
18     if (msg != NULL) {
19         PRINT(msg);
20     }
21
22 #ifdef WAIT
23 #ifdef __AVR__
24     while (sb_button_getState(BUTTON0) != PRESSED);
25 #else
26     getchar();
27 #endif
28 #endif
29 }
```

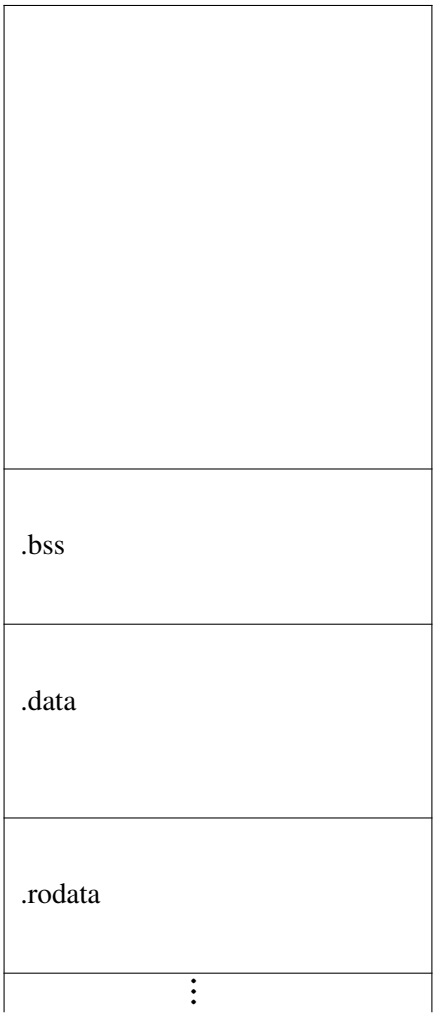
b) Lösen Sie auf der nachfolgenden Seite die C-Präprozessordirektiven für den C-Code der obenstehenden Datei **exam.c** vollständig auf. (5 Punkte)

Aufgabe 5: Speicherorganisation (9 Punkte)

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

```
static const char *text = "5PiC i5t c00l";
static const uint8_t BUFFER_SIZE = 3;

static volatile uint8_t move_text;

static void move_text_timer_callback(void) {
    move_text = 1;
}

void main(void) {
    sei();
    static uint8_t time = 400;
    sb_timer_setAlarm(
         move_text_timer_callback,
         time, time
    );

    const char *text_start = text;
    move_text = 1;

    while(42) {
        if(move_text){
            move_text = 0;
            if((*text_start) == '\0') {
                text_start = text;
            }

            char buffer[BUFFER_SIZE];
            buffer[0] = text_start[0];
            buffer[1] = text_start[1];
            buffer[2] = '\0';

            text_start++;
        }
        ...
    }
}
```

Speicheraufbau (vereinfacht):



a) Vervollständigen Sie den (vereinfachten) Speicheraufbau:

5 Punkte

1) Ergänzen Sie *Stack* und *Heap* sowie deren Wachstumsrichtung in der Abbildung. (2 Punkte)

2) Ordnen Sie alle im Quelltext vorkommenden Variablen den dargestellten Speichersektionen zu. (3 Punkte)

b) Die folgende Tabelle enthält vier Zeiger-Datentypen. Geben Sie für alle Datentypen an, ob der dereferenzierte Wert und der Zeiger veränderlich sind (jeweils **ja** oder **nein**). Sollte ein Datentyp so in C nicht möglich sein, kreuzen Sie bitte die Spalte **Syntaxfehler** an. (4 Punkte)

	Wert veränderlich	**Zeiger veränderlich**	**Syntaxfehler**
`uint8_t *`			
`const uint8_t *`			
`uint8_t * const`			
`const uint8_t * const`			

Aufgabe 6: Bitoperationen (9 Punkte)

Für diese Aufgabe ist `Bit 0` das niedrigstwertige Bit und `Bit 7` das höchstwertige Bit.

a) Kreuzen Sie an welche LEDs nach Aufruf von **`func(0x0f)`** leuchten.

2 Punkte

```
static void func(uint8_t leds) {
    sb_led_setMask(leds & 0x4c);
}
```

- ☐ `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 + 4) % 8)));
    }
}
```

- ☐ `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(0x1f)`** leuchten.

2 Punkte

```
static void func(uint8_t leds) {
    sb_led_setMask(leds ^ 0x55);
}
```

- ☐ `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 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*.

3 Punkte

```
static void func(void) {
    for(uint8_t i = 0; i < 6; i++) {
        sb_led_setMask(0xfe + i);
    }
}
```

Iteration 1 (i=0):

Iteration 2 (i=1):

Iteration 3 (i=2):

Iteration 4 (i=3):

Iteration 5 (i=4):

Iteration 6 (i=5):