Aufgabe 1: (20 Punkte)

Bei den Multiple-Choice-Fragen ist jeweils nur **eine** richtige Antwort eindeutig anzukreuzen. Auf die richtige Antwort gibt es die angegebene Punktzahl.

Wollen Sie eine Multiple-Choice-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 Aussage zur Speicherallokation ist richtig? 2 Punkte

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

b) Welche Aussage zu Variablen ist richtig? 2 Punkte

- ❏ Der Compiler sorgt dafür, dass Variablen vom Typ `int` immer eine geeignete Größe haben.
- ❏ Wenn man bei einer `int8_t`-Variable zum Wert `127` eins dazu addiert, hat sie den Wert `-128`.
- ❏ Eine `signed int`-Variable braucht mehr Speicher als eine `unsigned int`-Variable, weil zusätzlicher Platz für das Vorzeichen-Bit benötigt wird.
- ❏ Bei `char`-Variablen ist die Zuweisung von `'7'` gleichbedeutend mit der Zuweisung von `0x07`.

c) Gegeben ist folgender Ausdruck: 2 Punkte

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

Welche Aussage ist richtig?

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

d) Gegeben ist folgender Programmcode: 2 Punkte

```
#define SUB(a,b) a-b
#define MUL(a,b) a*b
```

Was ist das Ergebnis von folgendem Ausdruck:

```
4 * MUL( SUB(2,3), 4)
```

- ❏ -4
- ❏ -16
- ❏ 16
- ❏ 80

e) Welche Aussage zu globalen Variablen ist richtig? 2 Punkte

- ❏ Globale Variablen sind bei sehr großen Programmen vorteilhaft, weil man alle Variablendefinitionen an einer Stelle hinschreiben kann und man sie dadurch sehr schnell finden kann.
- ❏ Durch die Verwendung von globalen Variablen kann man den Einsatz von Funktionsparametern vermeiden. Dadurch werden Programme übersichtlicher und leichter wartbar.
- ❏ Man sollte globale Variablen sparsam einsetzen, da sie mehr Speicherplatz benötigen als lokale Variablen.
- ❏ Durch den Einsatz von globalen Variablen werden vor allem große Programme unübersichtlich und auf Dauer schwer wartbar, da der direkte Bezug zwischen Daten und den Funktionen verloren geht.

f) Was ist ein Stack-Frame? 2 Punkte

- ❏ Ein Bereich des Speichers, in dem lokale Variablen einer Funktion abgelegt sind.
- ❏ Ein bei Interrupts auftretendes spezielles Nebenläufigkeitsproblem.
- ❏ Ein spezieller Registersatz des Prozessors zur Bearbeitung von Funktionen.
- ❏ Ein Fehler, der bei unberechtigten Zugriffen auf den Stack-Speicher entsteht.

g) Welcher der folgenden Variablen benötigt genau 8 Byte Speicher? 2 Punkte

- ❏ `char a[] = "Hallo Du";`
- ❏ `char b[] = "Tollwut";`
- ❏ `struct {uint16_t a;  uint16_t b; uint16_t c;} c;`
- ❏ `union {uint32_t a; uint32_t b} d;`

h) Was bewirkt folgende Funktion? 2 Punkte

```
void func(int a, int b) {
      int c = a; a = b; b = c;
}
```

- ❑ Bei einem Aufruf vertauscht die Funktion die Inhalte der an die formalen Parameter a und b übergebenen Variablen.
- ❑ Da in C Funktionen mit "call by value" aufgerufen werden, erhält die Funktion Kopien der Aufrufparameter, die sie vertauscht. Beim Aufrufer hat dies allerdings keine Auswirkungen.
- ❑ Der Compiler meldet bei der Übersetzung einen Fehler, weil die Funktionsparameter nicht verändert werden dürfen.
- ❑ Die von b adressierte Speicherzelle enthält nach dem Aufruf die in der Variablen a abgelegte Speicheradresse.

i) Welche Aussage über den Begriff MMU ist richtig? 2 Punkte

- ❑ Die MMU rechnet Adressen des physikalischen Speichers in virtuelle Adressen um.
- ❑ Die MMU ist eine spezielle Einheit zur Verarbeitung von Mediendaten (Multi Media Unit), welche in den meisten Rechnern verbaut wird.
- ❑ Die MMU stellt sicher, dass Anwendungsprogramme keinen Zugriff auf den Speicher des Betriebssystems haben.
- ❑ Die MMU ist eine Softwarekomponente, die sich um die Zuweisung des Speichers an Prozesse kümmert (Memory Management Unit).

j) Was passiert, wenn man das folgende Programmstück übersetzen und ausführen möchte: 2 Punkte

```
char *string;
string = "SPiC ist toll";
```

- ❑ Der Compiler wird beim Übersetzen einen Fehler melden, weil diese Art von Zugriff auf einen Zeiger nicht erlaubt ist.
- ❑ Der Variablen `string` wird der Zeiger auf die konstante Zeichenkette "SPiC ist toll" zugeordnet.
- ❑ Unter Betriebssystemen wie Unix oder Windows wird zur Laufzeit eine Schutzverletzung bzw. SIGSEGV ausgelöst, da dem Zeiger ein Wert zugeordnet wird, der Zeiger jedoch nicht initialisiert wurde.
- ❑ Die Zeichenkette "SPiC ist toll" ist konstant und darf daher keinem nicht-konstanten Zeiger zugewiesen werden.

Aufgabe 2a: countdown (30 Punkte)

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

Schreiben Sie ein Countdown-Programm, welches nach Konfiguration eines Anfangswerts sekundenweise herunterzählt und schließlich das Auftreten des Alarms durch das Anzeigen einer LED-Alarmsequenz signalisiert. Ihr Programm soll wie folgt funktionieren:

- Während der Konfigurationsphase wird ein Countdown-Anfangswert durch das Potentiometer eingestellt, dessen Wert mit der Bibliotheksfunktion
  ```
  uint16_t poti_read(void);
  ```
  abgefragt werden kann. Die Funktion liefert einen 10-Bit-Wert zurück. Dessen Wertebereich soll gleichmäßig auf die Werte von 1 bis 8 abgebildet werden. Der gerade eingestellte Wert wird durch eine entsprechend gefüllte LED-Reihe angezeigt (z. B. Anfangswert 3: LEDs 0-2 leuchten).
- Wird der Taster gedrückt, wird der eingestellte Wert als Anfangswert gesetzt und die Countdown-Phase beginnt. Der aktuelle Zählerstand wird wieder durch den Füllstand der LEDs angezeigt. Vor jedem Countdown-Schritt wird durch eine Funktion
  ```
  void wait(void);
  ```
  für eine Sekunde gewartet. Gehen Sie vereinfachend davon aus, dass eine Warteschleife mit 60.000 Durchläufen einer Sekunde entspricht.
- Durch einen erneuten Tastendruck wird der Countdown abgebrochen. Dabei wird die aktuelle Wartesekunde noch zu Ende ausgeführt.
- Wenn der Countdown vollständig abgelaufen ist, wird eine LED-Alarmsequenz angezeigt: Dazu leuchten zunächst alle geradzahligen LEDs (0, 2, 4, 6) und danach alle ungeradzahligen LEDs (1, 3, 5, 7), gefolgt von je einer Wartesekunde. Bei einem Abbruch entfällt die LED-Alarmsequenz.
- Nach Abbruch oder Ablauf des Countdowns wird der Mikrocontroller in den Schlafmodus versetzt. Durch Tastendruck gelangt er wieder in die Konfigurationsphase.
- Es dürfen keine Annahmen über den initialen Zustand der Hardwareregister und über andere Interruptquellen getroffen werden. Die Initialisierung soll in einer Funktion `void init(void)` geschehen.

Information über die Hardware

LEDs: **PORTB**, Pins 0-7, aktiviert bei low-Pegel
- Pin als Ausgang konfigurieren: entsprechendes Bit in **DDRB**-Register auf 1

Taster: **PORTD**, Pin 2
- Pin als Eingang konfigurieren: entsprechendes Bit in **DDRD**-Register auf 0
- externe Interruptquelle **INT0**, ISR-Vektor-Makro: **INT0_vect**
- Aktivierung der Interruptquelle erfolgt durch Setzen des **INT0**-Bits im Register **GICR**.
- Taster verbindet den Pin mit Masse, es muss der interne Pullup-Widerstand verwendet werden (entsprechendes Bit in **PORTD**-Register auf 1 setzen).
- Konfiguration der externen Interruptquelle 0 (Bits in Register **MCUCR**)

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

/* Funktionsdeklarationen, globale Variablen, etc. */

/* Unterbrechungsbehandlungsfunktion */

/* Funktion main */

   /* Initialisierung und lokale Variablen */
```

A:


```
   /* Hauptschleife */

    /* Konfigurationsphase */

    /* Countdown-Phase */

    /*  LED-Alarm-Sequenz */
```

H:

```
    /* Schlaf-Phase */
```

```
} /* Ende Funktion main */

/* Initialisierungsfunktion */
```

```
/* Wartefunktion */
```

IW:

Aufgabe 2b: executor (18 Punkte)

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

Schreiben Sie ein Programm **`executor`**, welches die Dateinamen der Dateien im aktuellen Verzeichnis als Kommandos interpretiert und diese Kommandos nacheinander ausführt. Dabei sind der Programmname und die Parameter durch einen Unterstrich getrennt. Dateinamen, die mit einem "." beginnen, werden ignoriert!

Beispiel: Die Datei **`ls_opt`** führt **`ls`** mit dem Parameter **`opt`** aus.

Das Programm soll im Einzelnen wie folgt funktionieren:

- Nach dem Start soll das aktuelle Verzeichnis geöffnet werden und alle Dateien abgearbeitet werden. Eine besondere Reihenfolge muss hierbei nicht beachtet werden.
- Für jede Datei wird ein Kindprozess erzeugt, in welchem das Kommando ausgeführt wird.
- Vor Start des nächsten Kommandos wird auf das Ende der Bearbeitung gewartet, der Exit-Status muss nicht ausgegeben werden.
- Beim Auftreten eines Fehlers soll die Abarbeitung nach Möglichkeit fortgesetzt werden. Achten Sie auf sinnvolle Fehlermeldungen auf dem Standardfehlerausgabekanal. Sie können davon ausgehen, dass nie mehr als 126 Parameter übergeben werden.

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

```
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/stat.h>
#include <unistd.h>

/* Funktion main */
```

```
/* Verzeichnis oeffnen und durchsuchen */
```

D:


```
    /* gefundene Dateinamen in argv-Feld umwandeln */
```

```
    /* Kindprozess erzeugen, Programm starten */
```

```
    /* Fehlerbehandlung, Aufraeumen */
```

```
} /* Ende Funktion main */
```

E:

Aufgabe 3: (9 Punkte)

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

Gegeben sei folgendes C-Programm:

```
int a;
const int b = 4711;
volatile int c = 13;

void main(void)
{
        int d = 42;
        static int e;
        int *f;
        ...
}
```

a) Welche Speicherbereiche werden beim Übersetzen und Binden bzw. für die Ausführung eines C-Programms angelegt? Welche Arten von Daten befinden sich allgemein in diesen Bereichen? Ordnen Sie außerdem die obigen Variablen **a** bis **f** diesen Bereichen zu. (7 Punkte)

b) Wie verteilen sich die Speicherbereiche auf einem Mikrocontroller mit Harvard-Architektur auf RAM und Flash/ROM? (2 Punkte)

Aufgabe 4: (13 Punkte)

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

a) Was versteht man unter Nebenläufigkeit? Geben Sie eine kurze Definition des Begriffs. (2 Punkte)

b) In welchen Situationen tritt Nebenläufigkeit auf? (2 Punkte)

c) Bei Nebenläufigkeit können verschiedene Probleme auftreten. Beschreiben Sie exemplarisch zwei solche Probleme und wie man mit ihnen umgeht, um korrekt funktionierende Software zu gewährleisten. (5 Punkte)

d) Man unterscheidet bei Ausnahmesituationen zwischen Traps und Interrupts. Beschreiben Sie die beiden Arten und die wesentlichen Unterschiede. (4 Punkte)