Aufgabe 1: (18 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 Aussagen zum Schlüsselwort **`volatile`** sind richtig?

2 Punkte

- ☐ 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.
- ☐ Das Schlüsselwort **`volatile`** beschleunigt den Zugriff auf die damit definierte Variable.
- ☐ Das Schlüsselwort **`volatile`** erlaubt dem Compiler bessere Optimierungen durchzuführen.

b) Welche Aussage zum Präprozessor ist richtig?

2 Punkte

- ☐ Präprozessor-Makros sind ein adäquater Ersatz für C-Funktionen (z.B. **`#define`** `max(a,b) ((a > b) ? a : b)`)
- ☐ Leere Präprozessor-Makros sind nicht erlaubt (z.B. **`#define`** `USE_7SEG`)
- ☐ Präprozessor-Makros sind ausschliesslich in Header-Dateien (Dateiendung `.h`) erlaubt
- ☐ Präprozessor-Direktiven verändern den jeweiligen Quellcode vor dessen Übersetzung

c) Was versteht man unter Polling?

2 Punkte

- ☐ 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.
- ☐ Wenn ein Programm regelmäßig eine Peripherie-Schnittstelle abfragt, ob Daten oder Zustandsänderungen vorliegen.

d) Worin liegt der Unterschied zwischen den wie folgt in der Datei `test.c` deklarierten Variablen?

2 Punkte

```
const uint8_t foo = 23;
static uint8_t bar = 42;
```

- ☐ Der Wert der Variable `foo` kann (ohne den Einsatz von Zeigern) nur von Funktionen in der Datei `test.c` gelesen werden.
- ☐ Der Wert der Variable `bar` kann (ohne den Einsatz von Zeigern) nur von Funktionen in der Datei `test.c` geändert werden.
- ☐ Der Wert der Variable `foo` kann nur über einen Zeiger geändert werden.
- ☐ Der Wert der Variable `bar` kann nur über einen Zeiger geändert werden.

e) Welche Aussage zur Speicherallokation ist richtig?

2 Punkte

- ☐ `automatic`-Variablen werden im Heap allokiert.
- ☐ 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.

f) In welcher Situation kann ein lost-update-Problem auftreten?

2 Punkte

- ☐ Nur wenn eine Interrupt-Sperre zu lange gehalten wird.
- ☐ Wenn während einer Interrupt-Bearbeitung weitere Interrupts gesperrt sind.
- ☐ Bei der Modifikation von lokalen **`uint8_t`**-Variablen in zwei Interrupt-Handlern.
- ☐ Wenn sowohl in der main-Funktion als auch in einem Interrupt-Handler modifizierend auf die gleiche **`uint16_t`**-Variable zugegriffen wird.

g) Welchen Wert hat die Variable `x` nach Ausführung der folgenden Zeilen Code:

2 Punkte

```
uint16_t x = 7 ^ 3;
x &= 0xF;
```

- ☐ 4
- ☐ 7
- ☐ 15
- ☐ 343

h) Gegeben sei folgender Programmcode

2 Punkte

```
int limit = 5;
int max = 32 / (10 - 2 * limit);
```

Welche Aussage ist richtig?

- ☐ Das Ergebnis von `max` ist `0`
- ☐ Das Ergebnis von `max` ist `NaN` (keine gültige Zahl)
- ☐ Das Ergebnis von `max` ist `inf` (unendlich)
- ☐ Auf einem Linux-System wird das Programm mit einem Trap abgebrochen.

i) Welche Aussage zum Thema virtueller Adressraum ist richtig?

2 Punkte

- ☐ Virtuelle Adressen entsprechen Variablennamen in einem C-Programm. In Zeigern werden dagegen physikalische Adressen gespeichert, mit denen man die Abbildung umgehen kann.
- ☐ Die Abbildung von virtuellen auf physikalische Adressen erfolgt während der Programmlaufzeit durch eine spezielle Softwarekomponente
- ☐ Dieselbe virtuelle Adresse kann in verschiedenen Prozessen auf unterschiedliche physikalische Adressen abgebildet werden.
- ☐ Die Umrechnung von virtuellen zu physikalischen Adressen erfolgt beim Übersetzen durch den Compiler.

Aufgabe 2: Getränkeautomat (30 Punkte)

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

Für das Informatikhochhaus soll ein einfacher Verkaufsautomat für Erfrischungsgetränke entwickelt werden. Aus Kostengründen wird eine einfache Geldwechseleinheit verbaut, die lediglich 1-Euro-Münzen akzeptiert. Die Getränkeauswahl ist auf ein Produkt (die „FAU-Mate“) begrenzt und der Preis für eine Getränkeflasche beträgt 2 Euro.

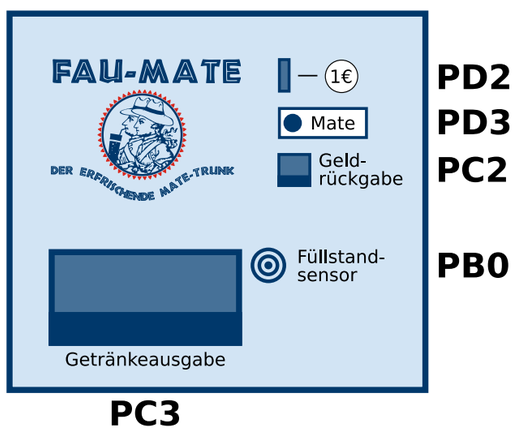
Schreiben Sie ein Programm für den AVR-Mikrocontroller, das den Getränkeautomaten steuert.

Ablauf:
Die bereits vorhandene elektronische Münzprüfeinheit validiert die eingeworfenen Geldstücke. Wird eine Münze nicht akzeptiert oder ist der Münzblock (= Geldkassette) bereits voll, dann wird die Münze automatisch wieder ausgegeben. Andernfalls wird sie dem Münzblock zugeführt und der Mikrocontroller per Interrupt über den Einwurf benachrichtigt.

Durch Drücken der Getränketaste wird eine Getränkeflasche ausgegeben, falls noch Flaschen vorrätig sind und der eingeworfene Geldbetrag ausreichend war. Wurde eine Fläche ausgegeben, wird der Restbetrag ausgezahlt. Andernfalls werden alle (zuvor eingeworfenen) Münzen wieder ausgegeben.

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.
- Implementieren Sie für die Ansteuerung von Getränkeausgabe und Geldrückgabe eine aktive Wartefunktion **`void`** `wait(`**`void`**`)`, die `WAITLOOPS` Schleifendurchläufe wartet.
- Jeder erfolgreiche Einwurf einer 1-Euro-Münze muss gezählt werden. Es darf davon ausgegangen werden, dass nie mehr als 200 Euro eingeworfen werden.
- Beim Drücken der Getränketaste wird geprüft, ob der Geldbetrag für ein Getränk ausreichend war (≥ 2 Euro) und Flaschen vorrätig sind. Sind beide Bedingungen erfüllt, wird der Auswurf **einer** Getränkeflasche initiiert.
- Anschließend wird der restliche Geldbetrag (im Fehlerfall der Gesamtbetrag) wieder ausgegeben.
- Sowohl Geldeinwurf als auch Tastendrücke sollen mit Hilfe von Interrupts erkannt werden. (kein Polling!)
- Während des Wartens auf Geldeinwurf oder Tastendrücke soll der Mikrocontroller zum Stromsparen in den Schlafmodus gehen.



Information über die Hardware

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

Geldeinwurf: Interruptleitung an **PORTD**, Pin 2
- active-low: Wird eine 1-Euro Münze angenommen, so liegt kurzzeitig ein LOW-Pegel an.
- Pin als Eingang konfigurieren: entsprechendes Bit in **DDRD**-Register auf 0
- Internen Pull-Up-Widerstand aktivieren; entsprechendes Bit in **PORTD**-Register auf 1
- Externe Interruptquelle **INT0**, ISR-Vektor-Makro: **INT0_vect**.
- Aktivierung der Interruptquelle erfolgt durch Setzen des **INT0**-Bits im Register **EIMSK**.

Getränketaste: Interruptleitung an **PORTD**, Pin 3
- active-low: Wird die Taste gedrückt, so liegt ein LOW-Pegel an.
- Pin als Eingang konfigurieren: entsprechendes Bit in **DDRD**-Register auf 0
- Internen Pull-Up-Widerstand aktivieren; entsprechendes Bit in **PORTD**-Register auf 1
- Externe Interruptquelle **INT1**, ISR-Vektor-Makro: **INT1_vect**.
- Aktivierung der Interruptquelle erfolgt durch Setzen des **INT1**-Bits im Register **EIMSK**.

Getränkefüllstand: **PORTB**, Pin 0
- Ist noch mindestens eine Flasche im Vorratsbehältnis, so liegt ein HIGH-Pegel an.
- Pin als Eingang konfigurieren: entsprechendes Bit in **DDRB**-Register auf 0
- Internen Pull-Up-Widerstand aktivieren; entsprechendes Bit in **PORTB**-Register auf 1
- Auslesen des Zustands über entsprechendes Bit in **PINB**-Register

Geldrückgabe: **PORTC**, Pin 2
- Ein HIGH-Pegel öffnet den Münzauswurf (es ist mechanisch sichergestellt, dass maximal eine 1-Euro-Münze ausgegeben wird).
- Der Münzauswurf muss nach einer kurzen Wartezeit (`WAITLOOPS` Iterationen sind ausreichend) wieder durch Anlegen eines LOW-Pegel geschlossen werden.
- Pin als Ausgang konfigurieren: entsprechendes Bit in **DDRC**-Register auf 1
- Münzauswurf zunächst geschlossen; entsprechendes Bit in **PORTC**-Register auf 0

Getränkeausgabe: **PORTC**, Pin 3
- Ein HIGH-Pegel öffnet die Flaschenauswurfklappe (es ist mechanisch sichergestellt, dass maximal eine Getränkeflasche ausgegeben wird).
- Die Klappe muss nach einer kurzen Wartezeit (`WAITLOOPS` Iterationen sind ausreichend) wieder durch Anlegen eines LOW-Pegel geschlossen werden.
- Pin als Ausgang konfigurieren: entsprechendes Bit in **DDRC**-Register auf 1
- Flaschenauswurfklappe zunächst geschlossen; entsprechendes Bit in **PORTC**-Register auf 0

Konfiguration der externen Interruptquellen **INT0** und **INT1** (Bits in 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

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>

#define WAITLOOPS 1000

/* Funktionsdeklarationen, globale Variablen, etc. */

/* Unterbrechungsbehandlungsfunktionen */
```

A:

```
/* Wartefunktion */

/* Ende Wartefunktion */

/* Funktion main */

   /* Initialisierung */

   /* Hauptschleife */

       /* Warten auf Ereignisse */
```

M:


```
       /* Getränkeausgabe und Geldrückgabe */

/* Ende main */
```

B:

```
/* Initialisierungsfunktion */
```

```
/* Ende Initialisierungsfunktion */
```

I:

Aufgabe 3: Programm zum parallelen Kopieren von Dateien (15 Punkte)

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

Schreiben Sie ein Programm `pcopy`, das mehrere Dateien gleichzeitig in ein angegebenes Zielverzeichnis kopiert. Der Aufruf des Programms erfolgt mit dem Zielverzeichnis als ersten Parameter und einer Liste von den zu kopierenden Quelldateien, zum Beispiel:

```
pcopy /data/ziel g1.dat mountain.SC76 final.wad
```

Das Programm soll im Detail wie folgt funktionieren:

- Das Programm prüft zu Beginn, ob mindestens zwei Parameter – das Zielverzeichnis sowie mindestens eine Quelldatei – übergeben wurde. Sollte dies nicht der Fall sein, gibt es eine entsprechende Fehlermeldung aus und beendet sich.
- Wurde das Programm korrekt aufgerufen, wird für jede Quelldatei mittels `fork()` eine Prozesskopie erstellt, in der die weitere Bearbeitung abläuft.
- Für jede Quelldatei wird in der zu implementierenden Funktion

  ```
  int copyFileIntoDir(const char *source, const char *targetDir);
  ```

  der Kopiervorgang der Quelldatei in den Zielordner initiiert. Dazu wird zuerst eine neue Zeichenkette mit dem Pfad `targetDir`, dem Verzeichnistrennsymbol `/` und dem Inhalt von `source` als Zieldatei generiert, anschließend wird diese an die Kopierfunktion `copyFile()` übergeben. Es darf davon ausgegangen werden, dass die Anzahl der Zeichen im Pfad zur Zieldatei immer kleiner als `PATH_MAX` ist.
- Die bereits vorgegebene Funktion

  ```
  int copyFile(const char *source, const char *target);
  ```

  prüft, ob die Quelldatei eine reguläre Datei ist und sich unterhalb des aktuellen Arbeitsverzeichnis befindet. Anschließend erstellt sie ggf. die benötigten Unterordner und kopiert den Inhalt von `source` nach `target`. Bei Erfolg wird `0` zurück gegeben, im Fehlerfall ein davon abweichender Fehlercode. Der Rückgabewert kann hierbei in Form des Exit-Status an den Hauptprozess durchgereicht werden.
- Der Hauptprozess wartet (nach dem Starten aller Kopierprozesse) mittels `wait()` auf den Abschluss der Kopierprozesse und wertet die Rückgabewerte aus. Im Anschluss soll eine Meldung (auf dem `stdout`-Kanal) die Kopiervorgänge zusammenfassen:

  ```
  2 von 3 Dateien erfolgreich kopiert
  ```

  Sofern nicht alle Dateien erfolgreich kopiert werden konnten, wird das Programm mit `EXIT_FAILURE` beendet.

Hinweise:

- Achten Sie auf eine korrekte Fehlerbehandlung der verwendeten Funktionen.
- Die Ausgabe von Fehlern soll (ggf. mit Hilfe der `errno`-Variable) auf dem `stderr`-Kanal erfolgen.

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

```c
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

#define PATH_MAX 4096

extern int copyFile(const char *source, const char *target);

static int getExitStatus(int status){
    if(WIFEXITED(status) != 0){
        return WEXITSTATUS(status);
    }
    return 1;
}
```

// Funktion copyFileIntoDir

// Ende copyFileIntoDir

H:

// Funktion main

// Parameter Fehlerbehandlung

// Kindprozesse erzeugen

F:

```
    // Auf Kindprozesse warten
```

```
    // Ausgabe
```

```
// Ende main
```

W:

Aufgabe 4: Speicherorganisation (10 Punkte)

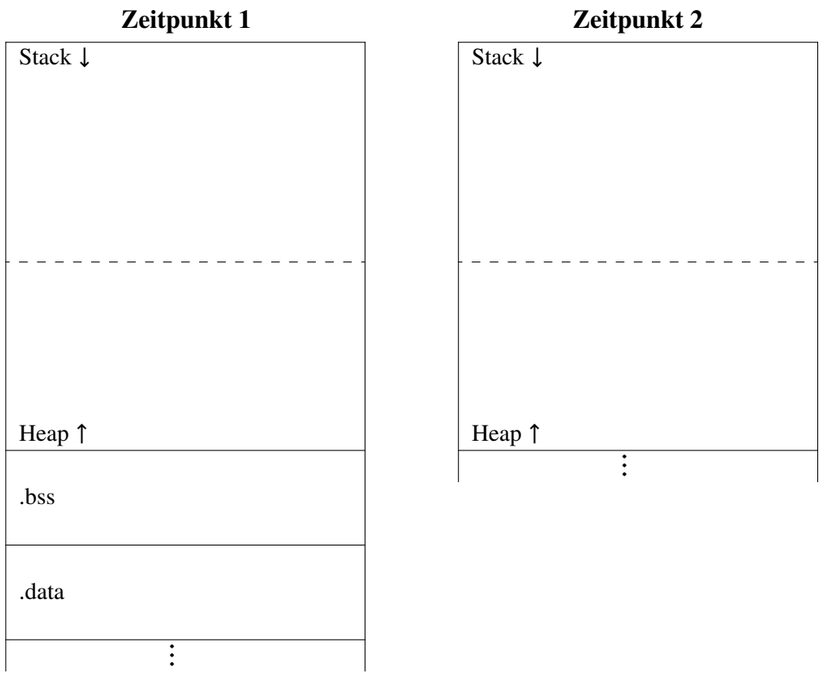
Sie dürfen diese Seite zur besseren Übersicht heraustrennen!

Das folgende (kompilierende) Programm wird auf einer IA32 Architektur ausgeführt und berechnet die auf die Zahl `23` folgende Primzahl.

Hinweis: Lesen Sie zuerst die Aufgabenstellung – ein vollständiges Verständnis der Primzahlenberechnung ist zur Bearbeitung der Aufgabe nicht notwendig!

```
#include <stdlib.h>
#include <stdio.h>
#define MAX 1000

int n;
int s = 2;

int checkPrime(int * p) {
    for (int i = 0; i <= n; i++)
        p[i] = 1;
    for (int i = s; i < n; i++)
        for (int j = i; i * j <= n; j++)
            p[i * j] *= 1 - p[i];
    return p[n];
}

int nextPrime(int ** k) {
    int z = 0;
    static int * a = NULL;

    // Zeitpunkt 1

    if (a == NULL)
        a = malloc(sizeof(int) * MAX);
    while (1) {
        n++;
        if (checkPrime(a)) {
            *k = &z;
            return n;
        }
        z++;
    }
}

void main() {
    n += 23;
    int * skip;
    int prime = nextPrime(&skip);

    // Zeitpunkt 2

    printf("Naechste Primzahl ist %d (%d uebersprungen)\n", prime, *skip);
}
```

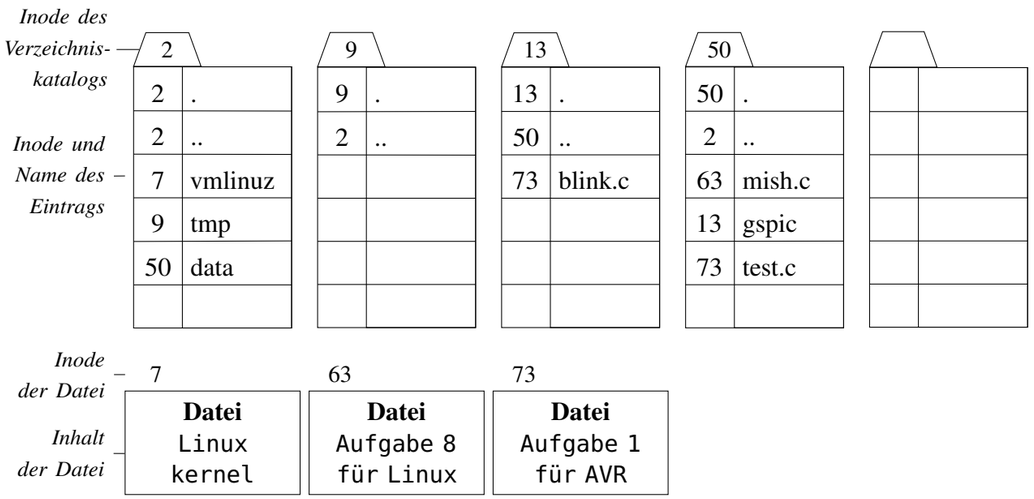
a) Vervollständigen Sie die folgenden beiden Abbildungen so, dass sie den den vereinfachten Speicheraufbau (RAM) während der Ausführung des obigen Programms zum Zeitpunkt 1 und Zeitpunkt 2 zeigen. Fügen Sie dazu die zu den zwei im Quellcode annotierten Zeitpunkten vorhandenen (*lebendigen*) Variablen an den entsprechenden Speicherbereichen in die Abbildungen ein (Variablenname ist ausreichend). Sollte zu den jeweiligen Zeitpunkten Speicher existieren, der dynamisch mit `malloc()` alloziert wurde, dann kennzeichnen Sie diesen durch den Buchstaben **M** im zugehörigen Speicherbereich. (8 Punkte)



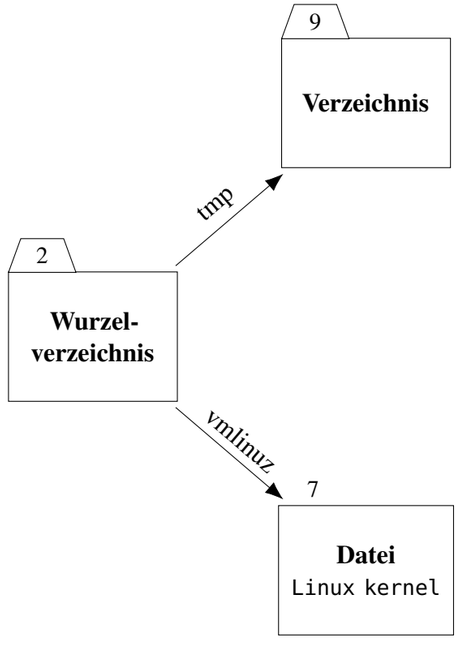
b) Welches Problem besteht bei der Dereferenzierung `*skip` in `main`? (2 Punkte)

Aufgabe 5: Dateisystem (9 Punkte)

Ein Dateisystem ermöglicht das strukturierte Ablegen von Daten. Nachfolgend ist ein beispielhafter und vereinfachter Ausschnitt der hierfür in Linux benötigten Verwaltungsinformationen abgebildet.



a) Vervollständigen Sie den dazugehörigen Verzeichnisbaum. Orientieren Sie sich dabei an dem bereits vorgegebenen Schema: Verzeichnisse und Dateien werden mit einem beschrifteten Rechteck gekennzeichnet, ein Verweis mit einem beschrifteten Pfeil. Das Einzeichnen der `.` und `..` Verweise ist allerdings **nicht** erforderlich. (4.5 Punkte)



b) Nun soll das Verzeichnis `/data/spic/` erstellt werden. Anschließend wird darin ein Verweis mit dem Namen `temp.c` auf die bereits existierende Datei mit der Inode-Nummer 63 erstellt. Vervollständigen Sie dazu die oben in der Angabe vorgegebenen Verwaltungsinformationen. Eine Anpassung des Verzeichnisbaums aus Teilaufgabe a) ist **nicht** erfoderlich. (2.5 Punkte)

c) Erklären Sie den Unterschied zwischen **Verweisen** (Hard Links) und **symbolischen Verweisen** (Symbolic Links) im Hinblick auf das Löschen von Dateien. (2 Punkte)

Aufgabe 6: Nebenläufigkeit (8 Punkte)

Im Rahmen des Safecast-Projekts wird ein Dosisleistungswarngerät (zur Messung der augenblicklichen radioaktiven Strahlendosis) auf Basis eines 8-Bit-AVR-Mikrocontrollers gebaut.

In dem Gerät wird bei ionisierender Strahlung aufgrund von elektromagnetischen Wechselwirkungen im Zählrohr ein Interrupt (**`INT0`**) ausgelöst. Die Anzahl der Interrupts (gezählt in `anzahl`) ist ein Maß für die radioaktive Strahlendosis.

Sobald die Anzahl in einem definierten Zeitintervall (eine Minute) eine vorgegebene Grenze (`LIMIT`) überschreitet, wird ein Alarm ausgelöst.

```
#define LIMIT 2500

extern void alarm();
extern void delay_sec(uint8_t time);
static volatile uint16_t anzahl = 0;

ISR(INT0_vect){

    anzahl++;

    if (anzahl > LIMIT)
        alarm();

}

void main(){
    /* ... */
    while (1){

        // 1 Minute warten
        delay_sec(60);

        // Zuruecksetzen
        anzahl = 0;

    }
}
```

a) Das Programm enthält ein Nebenläufigkeitsproblem. Benennen Sie dieses und kennzeichen 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) Benennen Sie eine Technik, mit der sich das Problem vermeiden lässt. Nehmen Sie die entsprechenden Änderungen am Programmcode vor. (2 Punkte)