Aufgabe 1: (18 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) Gegeben ist folgender Programmcode: 2 Punkte

```
uint16_t x[] = {1, 2, 3, 5, 7};
uint16_t *y = &x[4];
y--;
```

Welchen Wert liefert die Dereferenzierung von `y`?

- ❏ 2
- ❏ 5
- ❏ 4
- ❏ Zur Laufzeit tritt ein Fehler auf.

b) Welche der folgenden Aussagen bzgl. der Interruptsteuerung ist richtig? 2 Punkte

- ❏ Pegelgesteuerte Interrupts werden beim Wechsel des Pegels ausgelöst, daher der Name.
- ❏ Pegelgesteuerte Interrupts müssen durch Pollen des Pegels abgefragt werden.
- ❏ Wurde gerade ein Flanken-gesteuerter Interrupt ausgelöst, so muss erst ein Pegelwechsel der Interruptleitung stattfinden, damit erneut ein Interrupt ausgelöst werden kann.
- ❏ Interrupts sind eine Besonderheit von AVR-Mikroprozessoren. Auf anderen Architekturen müssen externe Ereignisse durch Pollen abgefragt werden.

c) Was versteht man unter den formalen und tatsächlichen Parametern einer Funktion? 2 Punkte

- ❏ Der tatsächliche Parameter enthält eine Kopie des Aufrufparameters, der formale Parameter ist ein Zeiger auf den Aufrufparameter.
- ❏ Die formalen Parameter sind die Namen, unter denen auf die Aufrufparameter innerhalb der Funktion zugegegriffen werden kann.
- ❏ Die tatsächlichen Parameter sind die Rückgabewerte eines Funktionsaufrufs.
- ❏ Die tatsächlichen Parameter sind die Namen, unter denen innerhalb einer Funktion auf die Aufrufparameter zugegriffen wird.

d) Welche Aussage zu Zeigern ist richtig? 2 Punkte

- ❏ Ein Zeiger darf nie auf seine eigene Speicheradresse verweisen.
- ❏ Der Speicherbedarf eines Zeigers ist unabhängig von der Größe des Objekts, auf das er zeigt.
- ❏ Ein Zeiger kann zur Manipulation von schreibgeschützten Datenbereichen verwendet werden.
- ❏ Zeiger vom Typ `(void *)` sind am besten für Zeigerarithmetik geeignet, da sie kompatibel zu jedem Zeigertyp sind.

e) Gegeben sei folgende Enumeration: 2 Punkte

```
enum FARBE {rot, gruen, blau};
```

Welche Aussage ist richtig?

- ❏ Der Compiler meldet einen Fehler, weil den enum-Elementen kein Wert zugewiesen wurde.
- ❏ Der Wert von `blau` ist 2.
- ❏ Der Wert von `blau` ist 3.
- ❏ Der Wert von `blau` ist unbekannt; der Compiler weist zur Übersetzungszeit jedem enum-Element einen zufälligen, aber eindeutigen Wert zu.

f) Gegeben sei folgender Programmcode: 2 Punkte

```
int array[10];
int i;
for (i = 1; i <= 10; i++) {
    array[i] = 0;
}
```

- ❏ Allen Elementen des Feldes `array` wird durch Ausführung des Programm-Codes der Wert 0 zugewiesen.
- ❏ Das Programm stürzt mit einem Segmentation-Fault ab.
- ❏ Das Programm könnte in eine Endlosschleife geraten.
- ❏ Der C-Compiler meldet beim Übersetzen einen Semantikfehler.

g) Welche der folgenden Aussagen über den C-Präprozessor ist richtig? 2 Punkte

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

h) Welche Aussage zu Semaphoren ist richtig? 2 Punkte

- ❏ Semaphore werden benutzt, um in kritischen Abschnitten Interrupts zu sperren und so den gleichzeitigen Zugriff auf gemeinsame Datenstrukturen zu verhindern.
- ❏ Eine P-Operation überprüft, ob der Semaphor den Wert 0 hat und erhöht ihn anschließend.
- ❏ Semaphore können genutzt werden, um gegenseitiges Warten zwischen Threads zu implementieren.
- ❏ Die V-Operation dekrementiert den Semaphor um 1 und deblockiert andere in einer V-Operation blockierte Prozesse.

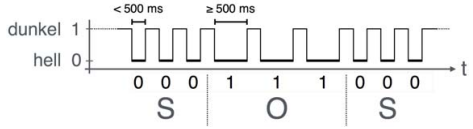
i) In Betriebssystemen wie Linux oder Windows unterscheidet man die Begriffe Programm und Prozess. Welche Aussage ist richtig? 2 Punkte

- ❏ Programme sind C-Quellcode-Dateien, die durch einen C-Compiler in einen lauffähigen Prozess übersetzt werden können.
- ❏ Prozesse können durch einen Aufruf der Funktion `exec(...)` terminiert werden.
- ❏ Ein Prozess kann mehrere Kindprogramme ausführen.
- ❏ Ein Prozess ist ein Programm in Ausführung

Aufgabe 2a: SOS-Erkennung (30 Punkte)

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

Schreiben Sie ein Programm für den AVR-Mikrocontroller, das Morsecode empfängt und das Notsignal "SOS" erkennt. Das Programm bekommt Interrupts eines Lichtsensors und bestimmt mit einem Timer die Länge der Impulse. Wird die Folge ···−−−··· erkannt, wird ein Alarm (blinkende LED) ausgelöst. Über einen Taster lässt sich das Gerät zurücksetzen.



Im Detail soll Ihr Programm wie folgt funktionieren:

- Initialisieren Sie die Hardware in der Funktion `void init(void)`. Treffen Sie keine Annahme über den initialen Zustand der Hardware-Register.
- Zu Beginn geht das Programm in den Empfangsmodus, in dem die Alarm-LED ausgeschaltet ist und der Mikrocontroller im stromsparenden Schlafmodus auf Ereignisse wartet.
- Eine fallende Flanke des Lichtsensors löst einen Interrupt aus, durch den die Zeitmessung beginnt. Beim nächsten Interrupt durch eine steigende Flanke wird durch erneute Zeitmessung die Dauer ermittelt. War die Dauer kürzer als 500 ms, wurde ein kurzes Zeichen erkannt, ansonsten ein langes Zeichen.
- Zur Zeitmessung steht Ihnen die Funktion `uint32_t getTime(void)` zur Verfügung, welche die Millisekunden seit Start des Mikrocontrollers zurückliefert. Diese Funktion darf nicht im Unterbrechungskontext aufgerufen werden.
- Die Zeichen sollen als Bits (0 = kurz, 1 = lang) mit einem Shift nach links in einer Bitmaske gespeichert werden. Wird das Notsignal durch den Vergleich der letzten 9 Bits mit dem vorgebenen Makro `SOS` erkannt, dann wechselt das Programm in den Alarmmodus.
- Im Alarmmodus blinkt die LED, indem sie alle 300 ms ihren Zustand ändert. Implementieren Sie für das Warten eine aktive Wartefunktion `void wait(uint32_t ms)`, die mit Hilfe der Funktion `getTime()` wartet, bis `ms` Millisekunden vergangen sind.
- Bei einem Tasterdruck wird die bislang empfangene Zeichenfolge zurückgesetzt. Befindet sich das Programm im Alarmmodus so wird außerdem in den Empfangsmodus zurückgewechselt.

Information über die Hardware

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

LED für Alarm: **PORTB**, Pin 5
- active-low: Die LED leuchtet sobald ein LOW-Pegel anliegt.
- Pin als Ausgang konfigurieren: entsprechendes Bit in **DDRB**-Register auf 1
- LED zu Beginn ausschalten; entsprechendes Bit in **PORTB**-Register auf 1

Taster: Interruptleitung **INT1** an **PORTD**, Pin 3
- active-low: Wird der Taster 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 **GICR**.

Lichtsensor: Interruptleitung **INT0** an **PORTD**, Pin 2
- active-low: bei eintreffendem Lichtstrahl 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 **INT0**, ISR-Vektor-Makro: **INT0_vect**.
- Aktivierung der Interruptquelle erfolgt durch Setzen des **INT0**-Bits im Register **GICR**.
- Abfrage des Pegels durch entsprechendes Bit in **PIND**-Register
  (0 = Licht; 1 = kein Licht)

Konfiguration der externen Interruptquellen 0 und 1 (Bits in Register **MCUCR**)

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 SHORT 0
#define LONG  1

#define SOS     0x0038 /* binary: ...000111000 */

/* Bibliotheksfunktionen */
uint32_t getTime(void);             // nicht aus ISR aufrufen!


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

```
/* Unterbrechungsbehandlungsfunktionen */
```

A:

```
/* Funktion main */

   /*  Initialisierung und lokale Variablen */

  /* Hauptschleife */

      /*  Warten auf Ereignisse */
```

K:


```
      /*  Ereignisse bearbeiten */

/* Ende main */
```

E:

```
/* Funktion wait */
```

```
/* Ende wait */
```

```
/* Initialisierungsfunktion */
```

WI:

Aufgabe 2b: Funktion simple_system und Shell (17 Punkte)

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

Teil 1:

Implementieren Sie eine vereinfachte Fassung der Bibliotheksfunktion `system()`

```
int simple_system(const char *command);
```

welche Kommandos in der System-Shell ausführt und auf deren Beendigung wartet.

Die **`simple_system()`**-Funktion erwartet Kommandos in Form einer Zeichenfolge (z. B. "ls -al" oder "sleep 100"). Sie erzeugt einen Kindprozess und startet dort die Ausführung einer Shell mit der übergebenen Zeichenfolge:

```
/bin/sh -c command
```

Anschließend wartet **`simple_system()`** bis der Kindprozess beendet wurde und gibt die **`status`**-Information (vgl. **`waitpid()`**) über den Kindprozess als Rückgabewert an den Aufrufer.

Ist die Ausführung der Shell nicht möglich, so beendet sich der Kindprozess mit dem Exitstatus 127.

Bei allen anderen Fehlern soll der Aufruf mit -1 zurückkehren.

Teil 2:

Erweitern Sie nun Ihre Lösung zu einer minimalen Shell, die eingegebene Kommandos (ohne Zeilenumbruch) in Aufrufe an `simple_system()` umsetzt.

Eingegebene Leerzeilen sollen ignoriert werden.

Bevor die Shell auf eine neue Benutzereingabe wartet, soll als Prompt "sish >" ausgeben werden.

Nach dem Aufruf von **`simple_system()`** soll mit Hilfe der (vorgegebenen) Funktion

```
void print_status(const char *command, int status);
```

das Kommando und der Exit-Status des Kindprozesses bzw. der aufgetretene Fehler ausgegeben werden.

Weitere Hinweise:

- Sie können davon ausgehen, dass eingegebene Befehlszeilen nie länger als 1024 Zeichen sind.
- Die Funktion simple_system() soll nur den Kindprozess einsammeln, der durch ihren Aufruf erzeugt wurde.
- simple_system() macht keine Ausgaben auf stdout oder stderr, sondern meldet Fehler nur über den Rückgabewert an den Aufrufer.
- Eine Behandlung von Signalen ist in Ihrer Implementierung nicht erforderlich.

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

```
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>

#define MAX_LINE (1024)
```

// Funktion simple_system

// Funktion print_status

S:


```
void print_status(const char *command, int status){
   if(status == -1 || (WIFEXITED(status) && WEXITSTATUS(status) == 127)){
      printf("%s: error\n", command); //perror(command);
   } else if (WIFEXITED(status)) {
      printf("%s: exit status %d\n", command, WEXITSTATUS(status));
   }
}
```

// Funktion main

// Ende main

M:

Aufgabe 3: Speicher (10 Punkte)

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

Die Abbildung zeigt den vereinfachten Speicheraufbau (RAM und ROM) des folgenden Programms während der Ausführung der Endlosschleife.

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

b) Zeichnen Sie den mit Hilfe von malloc() dynamisch allozierten Speicher ein und kennzeichnen Sie ihn mit dem Buchstaben M. (1 Punkt)

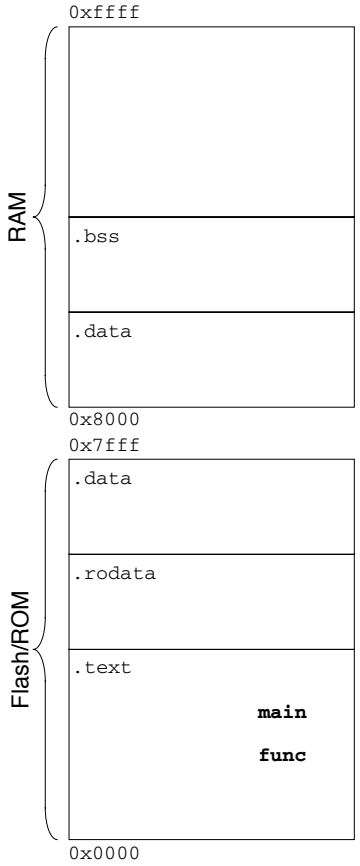
c) Ordnen Sie die Variablen a, b, c, d, s, x und p und ggf. ihre Initialisierungswerte den dargestellten Speicherbereichen zu und kennzeichnen Sie für Zeigervariablen den referenzierten Speicherbereich mit einem Pfeil. (5 Punkte)

```
int a = 5;
int b;
const int c = 42;
static int d = 0;

int func() {
   static int s = 3;
   return s++;
}

void main(void){
   int x = func();
   char *p = malloc(100);


   while(1){
     // wait forever
   }
}
```



d) Wie musste der RAM-Speicher beim Start des Mikrocontrollers vorbereitet werden, ehe die eigentliche Programmausführung in main() beginnen konnte? (2 Punkte)

Aufgabe 4: Synchronisation (6 Punkte)

Ein großer Versandhändler erfasst die Anzahl der Pakete, die minütlich das Versandzentrum verlassen, mit folgendem Programmfragment:

```
static volatile uint8_t packets = 0;

// Interruptbehandlung
ISR(INT0_vect){
    // Lichtschranke hat Paket erkannt
    packets++;
}

// Hauptprogramm
void main(void){

    /* ... */

    while(1) {

        wait_seconds(60);

        uint8_t count = packets;

        packets = 0;

        // Sende Paketanzahl der letzten 60 Sekunden
        transmit(count);

    }
}
```

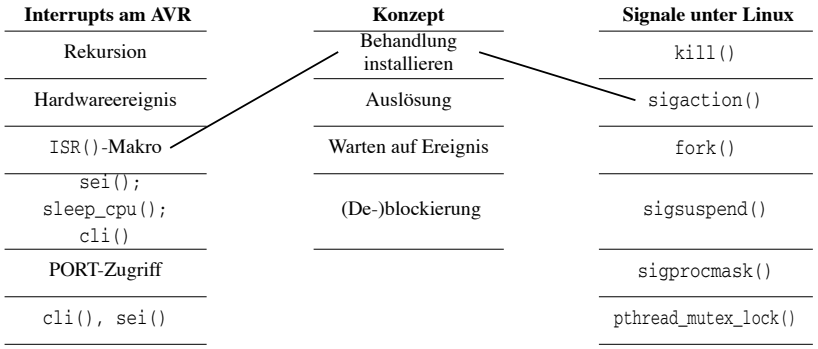
a) Das Programm enthält ein Nebenläufigkeitsproblem. Benennen Sie dieses und kennzeichen Sie den kritischen Abschnitt im Quellcode. (2 Punkte)

b) Skizzieren Sie einen konkreten Ablauf, der hier zu einem Nebenläufigkeitsfehler führt, und seine Auswirkungen (Stichpunkte, kurze Sätze). (2 Punkte)

c) Benennen Sie eine Technik, mit der sich das Problem vermeiden lässt. Nehmen Sie die entsprechenden Änderungen am Programmcode vor. (2 Punkte)

Aufgabe 5: (9 Punkte)

a) Sie haben asynchrone Signale unter Linux als Analogie zu Interrupts kennen gelernt. Ordnen Sie den gegebenen Konzepten den jeweils passenden Begriff auf der linken und der rechten Seite zu (analog zu dem Beispiel bei "Behandlung installieren"). (3 Punkte):



Interrupts am AVR	Konzept	Signale unter Linux
Rekursion	Behandlung installieren	`kill()`
Hardwareereignis	Auslösung	`sigaction()`
`ISR()`-Makro	Warten auf Ereignis	`fork()`
`sei(); sleep_cpu(); cli()`	(De-)blockierung	`sigsuspend()`
PORT-Zugriff		`sigprocmask()`
`cli(), sei()`		`pthread_mutex_lock()`

b) Nennen Sie zwei zur Programmausführung asynchrone Ereignisse, die zu einem Signal führen können. (2 Punkte)

c) Nennen Sie zwei zur Programmausführung synchrone Ereignisse (Traps), die zu einem Signal führen können. (2 Punkte)

d) Erläutern Sie kurz den Unterschied zwischen einem spin lock und einem sleeping lock und ihre Vor- bzw. Nachteile. (2 Punkte)