Problem 1: (12 Points)

For the single-choice questions in this problem, only **one** correct answer must be clearly marked with a cross. The correct answer is awarded the indicated number of points.

If you want to correct an answer, please cross out the incorrect answer with three horizontal lines (☒) and mark the correct answer with a cross.

Read the question carefully before you answer.

a) Given the following program code. What does the program output? 2 Points

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

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

- ☐ The C compiler reports an error during compilation.
- ☐ The program outputs `no match`.
- ☐ The program crashes at runtime.
- ☐ The program outputs `match`.

b) The following program code is given: 2 Points

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

What value does the dereferencing of `y` (i.e., `*y`) return after the program code has been executed?

- ☐ 8
- ☐ 1
- ☐ An error occurs at runtime.
- ☐ 4

c) The following expression is given: 2 Points

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

Which statement is correct?

- ☐ The initial value of `a` has no influence on the result.
- ☐ If `a` contains the value `5` and `b` contains the value `7`, it returns false.
- ☐ The compiler reports an error because this expression is not allowed.
- ☐ If `a` contains the value `7` and `b` contains the value `5`, it returns false.

d) How to solve the concurrency problem “lost update” between the main function and an interrupt handler on a microcontroller?

2 Points

- ☐ Using the keyword `volatile` solves all concurrency problems.
- ☐ By calling a callback function in the interrupt handler.
- ☐ Through synchronization by temporarily disabling of the interrupts.
- ☐ Through the use of level-trigger instead of edge-triggered interrupts.

e) Given the following program fragment for an AVR microcontroller:

2 Points

```
uint8_t a = 100;
uint8_t b;

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

Which of the following statements is correct?

- ☐ `b` has the value `350` after executing the assignment.
- ☐ `b` has the value `250` after the assignment has been executed.
- ☐ The compiler warns of an interger overflow at compile time.
- ☐ During execution, an interger overflow occurs; however, this remains undetected on the AVR platform.

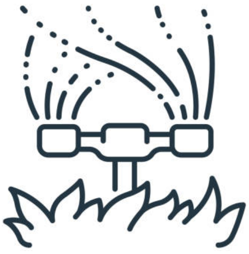
f) Which statement on the term “polling” is correct?

2 Points

- ☐ When a program periodically polls a peripheral interface for data or state changes.
- ☐ When a device triggers interrupts until the data has been fetched by the microcontroller.
- ☐ When a device requests data from a microcontroller by triggering an interrupt.
- ☐ When a program blocks interrupts to access critical data.

Problem 2: Irrigation System (30 Points)

You may detach this page for a better overview during programming!



Implement the control of an automatic irrigation system that automatically waters plants. Operation should be as simple as possible: the button wakes the system from a deep sleep, the light sensor measures the brightness, and calculates a watering time based on the brightness. Once the watering time has been calculated, it is shown on the 7-segment display and the water is switched on for the calculated time in minutes. During watering, the blue indicator light (`BLUE0`) should be switched on and the 7-segment display should show the remaining time. After the previously calculated time has elapsed, the system should automatically return to sleep mode and wait for the next input.

The program should work in detail as follows:

- Initialize the hardware in the function `void init(void)`. Do not make any assumptions about the initial state of the hardware registers.
- The input `PD2` (interrupt 0) is connected to the button. A falling edge occurs exactly when the button is pressed and a rising edge occurs when it is released again. You can assume that the button is not initially held down.
- An 8-bit timer should be used for timing. Configure the most resource-efficient prescaler and trigger an event once per second. You will find details on the next page.
- When the button is pressed, the brightness is to be determined by calling `int16_t sb_adc_read(ADCDEV dev)` for the ADC device `PHOTO`. This function returns an unsigned 10-bit integer value. The interrupts must be disabled during its invocation. Calculate a value between 3 minutes for minimum brightness and 14 minutes for maximum brightness in seconds and return it. The following note will help you.
- **Note**: The value range of the measured brightness is `0` to `1023`, so you could perform an integer devision of the measured value by `100` to map the brightness to an interval $[0, 11]$. You can then add a suitable offset to this normalized value to map it to the target interval.
- Now show the calculated duration in minutes on the 7-segment display.
- Set the output `PD7` so that the blue status LED `BLUE0` lights up to signal ongoing operation.
- Use the predefined functions `void watering_on(void)` and `void watering_off(void)` from the file `watering.h` to switch the watering on and off.
- Use the function `uint8_t sb_7seg_showNumber(uint8_t)` to display the watering time on the 7-segment display and `void sb_7seg_disable(void)` to switch the 7-segment display off again afterwards.
- After the watering time has elapsed, the water, 7-segment display, and indicator light should be switched off and the microcontroller goes back into sleep mode until the button is pressed again.

Information about the hardware

You may detach this page for a better overview during programming!

Button: interrupt line to **PORTD**, pin 2

- Falling edge: button is pressed
- Rising edge: button is released
- Configure pin as input: corresponding bit in the **DDRD** register to 0
- Activate internal pull-up resistor: corresponding bit in the **PORTD** register to 1
- External interrupt source **INT0**, ISR vector macro: **INT0_vect**
- Activating/deactivating the interrupt source is done by setting/clearing the **INT0** bit in the **EIMSK** register

Configuration of the external interrupt source **INT0** (bits in **EICRA** register)

interrupt 0		description
ISC01	ISC00	
0	0	interrupt at low level
0	1	interrupt on either edge
1	0	interrupt on falling edge
1	1	interrupt on rising edge

Operating LED: output at **PORTD**, pin 7

- Shows that watering is currently in process.
- Configure pin as output: set the corresponding bit in the **DDRD** register to 1
- Initially switch off indicator light, corresponding bit in **PORTD** register to 1

Timer (8-bit): **TIMER0**

- The overflow interruption is to be used (ISR vector macro: **TIMER0_OVF_vect**)
- The most resource-efficient prescaler (*prescaler*) is 64, which causes the 8-bit counter **TCNT0** to overflow every $1ms$ at the 16MHz CPU clock (sufficiently accurate).
- Activating/deactivating the interrupt source is done by setting/clearing the **TOIE0** bit in the register **TIMSK0**

Configuration of the frequency of the timer **TIMER0** (bits in register **TCCR0B**)

CS02	CS01	CS00	description
0	0	0	timer off
0	0	1	CPU clock
0	1	0	CPU clock / 8
0	1	1	CPU clock / 64
1	0	0	CPU clock / 256
1	0	1	CPU clock / 1024
1	1	0	Ext. clock (falling edge)
1	1	1	Ext. clock (rising edge)

Complete the following code skeleton so that a fully compilable program is created.

```
#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:

Problem 3: lazy (21 Points)

Recurring tasks, such as the handing out assignments, automatic plagiarism checks or corrections, are an integral part of the exercises. To further automate these tasks, implement a program `lazy` that executes a given command repeatedly. The approximate waiting time between command invocations should also be passed as a command-line argument.

```
# Execute the command "plagiatscheck.sh blink" every 300 seconds
$> ./lazy 300 plagiatscheck.sh blink
```

The program should work in detail as follows:

- At the beginning, the program checks whether at least two parameters have been passed. If this is not the case, it issues a corresponding error message and exits.
- If the program has been called correctly, the required handling routines for the signals `SIGALRM` and `SIGCHLD` should be registered using `sigaction`. Each of them should set a corresponding event variable.
- The external auxiliary functions `parse_pos_integer` should be used to parse the passed waiting time.
- The parsed waiting time should then be passed to the external function `start_timer`. It starts a periodic timer that sends a `SIGALRM` to the calling process after the interval has expired.
- Now wait passively using `sigsuspend` until at least one of the two event variables has been set.
- If a `SIGALRM` has been delivered in the meantime, create a new child process (`fork`) and execute the passed program with its arguments (`exec`). In case of an error, only a appropriate warning should be displayed and the main program *not* should be terminated.
- If child processes have been terminated in the meantime and this has been signaled accordingly, their used process resources should be freed in a loop using `waitpid`.

Ensure correct error handling of the functions used. Error messages should generally be sent to `stderr`.

Complete the following code skeleton so that a fully compilable program is created.

```
#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:

Problem 4: System-Programming Language C (9 Points)

a) Consider the following code snippet. Answer the following questions in bullet points:

1. What is the meaning of the keyword **static** in line 1?
2. What is the meaning of **static** in line 2?
3. What is the meaning of the keyword **volatile** in example code?
4. Explain why we use the fixed-width types defined in `stdint.h` (e.g. `uint16_t`) for microcontroller programming.

(4 Points)

```
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 }
```

You may detach this page for a better overview during programming!

print.h:

```c
#ifdef NO_PRINT
#define PRINT(msg)
#else
#define PRINT(msg) printf(msg)
#endif

void printf(const char* msg);
```

exam.c:

```c
#define WAIT
#define __AVR__
#define NO_PRINT

#include "print.h"

#ifdef __AVR__
void main(void) {
    sei();
    pointerDemo();
#else
int main(void) {
    return pointerDemo();
#endif
}

static void wait_msg(const char *msg) {
    if (msg != NULL) {
        PRINT(msg);
    }

#ifdef WAIT
#ifdef __AVR__
    while (sb_button_getState(BUTTON0) != PRESSED);
#else
    getchar();
#endif
#endif
}
```

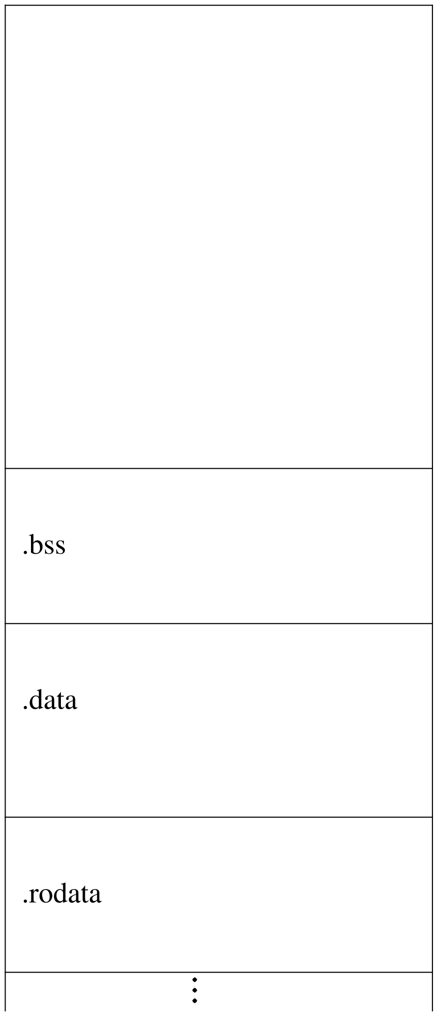
b) On the following page, completely expand the C-preprocessor directives for the C code of the file **exam.c** above. (5 Points)

Problem 5: Memory Layout (9 Points)

Note: Read the task first - a complete understanding of the program is not necessary.

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

Memory Layout (simplified):



a) Complete the (simplified) memory layout: 5 Points

1) Add the terms *stack* and *heap* and their direction of growth in the figure. (2 Points)

2) Assign all variables occurring in the source code to the corresponding memory segments. (3 Points)

b) The following table contains four pointer types. For all types, specify whether the dereferenced value and the pointer are mutable (i.e., not constant) (**yes** or **no** in each case). If a data type is not possible in C, please check the column **syntax error**. (4 Points)

	Value mutable	**Pointer mutable**	**Syntax error**
`uint8_t *`			
`const uint8_t *`			
`uint8_t * const`			
`const uint8_t * const`			

Problem 6: Bit Operations (9 Points)

For this problem, `bit 0` is the least significant bit and `bit 7` is the most significant bit.

a) Check which LEDs light up after calling **`func(0x0f)`**.

2 Points

```
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)

Notes:

b) Check which LEDs light up after calling **`func(4)`**.

2 Points

```
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)

Notes:

c) Check which LEDs light up after calling **`func(0x1f)`**.

2 Points

```
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)

Notes:

d) Describe which LEDs light up for each iteration when calling `func()`. You do not have to name any specific LEDs. Example answers: *the top four LEDs*, *all LEDs for which a 1 is set in* `leds`, *the bottom LED*.

3 Points

```
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):