Problem 1: (14 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) 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 program blocks interrupts to access critical data.
- ☐ When a device triggers interrupts until the data has been fetched by the micro-controller.
- ☐ When a device requests data from a microcontroller by triggering an interrupt.

b) When can concurrency problems occur? 2 Points

- ☐ If the program sections in the `if` and `else` part of a conditional statement access the same variable.
- ☐ If the same global variable is written from the main program and an interrupt service routine.
- ☐ If a program uses multiple local variables in a function.
- ☐ If a program section is run several times in a loop.

c) The following expression is given: 2 Points

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

Which statement is correct?

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

d) Given the following enumeration: `enum FARBE {red, green, blue;}` 2 Points

Which statement is correct?

- ☐ The value of `blue` is unknown; the compiler assigns a random but unique value to each `enum` element at translation time.
- ☐ The value of `blue` is 2.
- ☐ The value of `blue` is 3.
- ☐ The compiler reports an error because no value has been assigned to the enum elements.

e) The following macro definition can be found in the AVR library:

2 Points

```
#define PINA (*(volatile uint8_t *)0x39)
```

Which of the following statements regarding the use of the `volatile` keyword is correct in this case?

- ☐ If port A is configured as an input, the value of `PINA` could change at any time. `volatile` instructs the compiler to always read the current value from `PINA`.
- ☐ The `volatile` keyword enables safe access to individual bits of the register.
- ☐ The keyword `volatile` allows the compiler to perform better optimizations.
- ☐ The `volatile` keyword ensures that access to `PINA` is synchronized with interrupts.

f) Given are the following macros:

2 Points

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

What is the result of the following expression

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

- ☐ -16
- ☐ -4
- ☐ 80
- ☐ 16

g) 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?

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

Problem 2: Safe (29 Points)

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

Implement the control logic of a digital safe with a numerical combination lock. The safe is operated by a potentiometer for number input and a button for confirmation. The user turns the potentiometer to set a number (0-99), presses the button to confirm, and repeats this three times to enter a total of three numbers. If the correct combination has been entered (stored in the array `combination[]`), the safe opens. The current potentiometer value is updated on the 7-segment display every 10ms.

The details of the program are 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. A rising edge occurs when it is released again. You can assume that, initially, the button is not pressed down.
- An 8-bit timer should be used for the timing. Configure the most resource-efficient prescaler and trigger an event every 1ms. You will find details for this on the next page.
- When the configured count value of the timer is reached, an internal counter has to be incremented in `ISR(TIMER0_OVF_vect)`. A timer event should be signalled for every 10th overflow (every 10ms).
- When a timer event occurs, the current potentiometer value should be read with `sb_adc_read(POTI)`. Interrupts must be disabled during the call. Convert the 10-bit ADC value (0-1023) to a display value (0-99) and show it on the 7-segment display.
  Note: The ADC value range is 0 to 1023. To map this to the range of 0-99, you can multiply the ADC value by 100 and then divide it by 1024: `(adc_value * 100) / 1024`.
- When the button is pressed (button event), the current potentiometer value is stored as input for the current position of the combination. After the third input, check if the entered combination is correct.
- If the combination is correct, the function `open_safe()` has to be called. Then, reset the system for a new input.
- Ensure that the microcontroller is in sleep mode as long as possible and only wakes up on events. Remember that the sleep enable bit must be set with `sleep_enable` before `sleep_cpu` can be used.

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: set 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 on low level
0	1	interrupt on either edge
1	0	interrupt on falling edge
1	1	interrupt on rising edge

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

- The overflow interrupt has 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 <7seg.h>

extern int16_t sb_adc_read(ADCDEV dev);
extern int8_t sb_7seg_showNumber(int8_t nmbr);
extern void open_safe(void);

static uint8_t combination[3] = {23, 45, 67};

// Function declarations, global variables, etc.







// End function declarations, global variables, etc.


// Interrupt service routines














// End interrupt service routines
```

D:

```
// Main function

    // Initialization and declaration of local variables

    uint8_t input[3] = {0, 0, 0};

    // Main loop

        // Process timer event for display update
        // Process button event for number input











        // Check combination if three numbers have been entered
























// End main
```

M:

```
// Initialization function
```

```
// End initialization function
```

I:

Problem 3: `pot` (19 Points)

An overview of all running processes on the system often forms the first step in diagnosing problems. Implement a program `pot` (process overview table) that lists all running processes together with the most important allocated resources in tabular form.

```
$ ./pot
|  PID |    %CPU |    %MEM |      TIME |
|------|---------|---------|-----------|
|   42 |  20.25% |   7.50% |  12:34:56 |
| 4096 |   3.14% |  40.96% |  78:90:12 |
```

The program should work in detail as follows:

- The program initially outputs the header line with all corresponding column identifiers (`PID`, `%CPU`, `%MEM`, ...) in the correct format by using the provided helper function `print_prologue` (see next page for function definition).
- Next, the program opens the directory `/proc/` (see `PROC` macro) and iterates over its entries. Use the helper function `int check_proc_entry(const char *name)` to check all entries. This function, which you have to implement by yourself, checks whether the passed filename `name` consists exclusively of the digits `0` to `9`, which corresponds to the PID of the respective process. In case of success, `0` should be returned, otherwise `-1`. If a filename is invalid, ignore the filename and continue with the next entry. A possible directory structure for `/proc/` looks as follows:

```
$ ls -l /proc/
dr-xr-xr-x  root  root  .
dr-xr-xr-x  root  root  ..
dr-xr-xr-x  root  root  42
dr-xr-xr-x  root  root  4096
```

- Next, dynamically allocate a buffer of minimal size to be able to compose the absolute file path in the format `/proc/<PID>/stat`. Open the file at the composed filename and read its content. You can assume that the file content consists of at most 255 characters (excluding `'\0'`). You can use a buffer of size `MAX_FILE_LEN` on the stack for this purpose. A potential file content for example `/proc/42/stat` may look as follows:

```
$ cat /proc/42/stat
20.25%;7.50%;12:34:56
```

- The line read from the file then has to be split into the above-mentioned resources using the separators `";\n"`. The individual substrings should be passed together with the filename (PID) to the (provided) helper function `print_line` (for description, see next page) for output in the correct format. After that, the next entry should be processed, and output again if applicable.

Ensure correct error handling of the used functions. Error messages should generally be sent to `stderr`. The program should only be terminated in the case of unrecoverable errors (e.g. out of memory), otherwise - especially in context of errors related to individual files - an error message should be displayed and the program should continue with the next entry.

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

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

#define PROC "/proc/"
#define MAX_FILE_LEN 256

// Prints prologue of top command.
// Outputs the following line correctly formatted:
// |  PID |    %CPU |    %MEM |      TIME |
// |------|---------|---------|-----------|
extern void print_prologue(void);
// Prints single line of top command.
// Outputs PID, CPU usage, memory utilization, and total execution time, e.g.:
// |   42 |  20.25% |   7.50% |  12:34:56 |
extern void print_line(char *pid, char *cpu, char *mem, char *time);
// Prints system error message and exits with an error.
static void die(const char *message) {
  perror(message);
  exit(EXIT_FAILURE);
}
// Prints system error message.
static void warn(const char *message) {
  perror(message);
}

// Function check_proc_entry
```

C:

```
// Function main

  // Print header row

  // Open directory for reading

  // Iterate over directories

    // Check file name of entry

    // Allocate buffer for path for "/proc/<pid>/stat"

    // Concat path
    // Open path for reading







    // Read file










    // Parse file




    // Print line

    // Release resources






  // Close directory
```

M:

Problem 4: Memory Layout (9 Points)

Note: Read the problem description first. A complete understanding of the program is not necessary to complete the problem. If memory values are undefined, mark them with a **?** (e.g., *z = ?*, the value is z is unknown).

a) Complete the (simplified) memory layout:

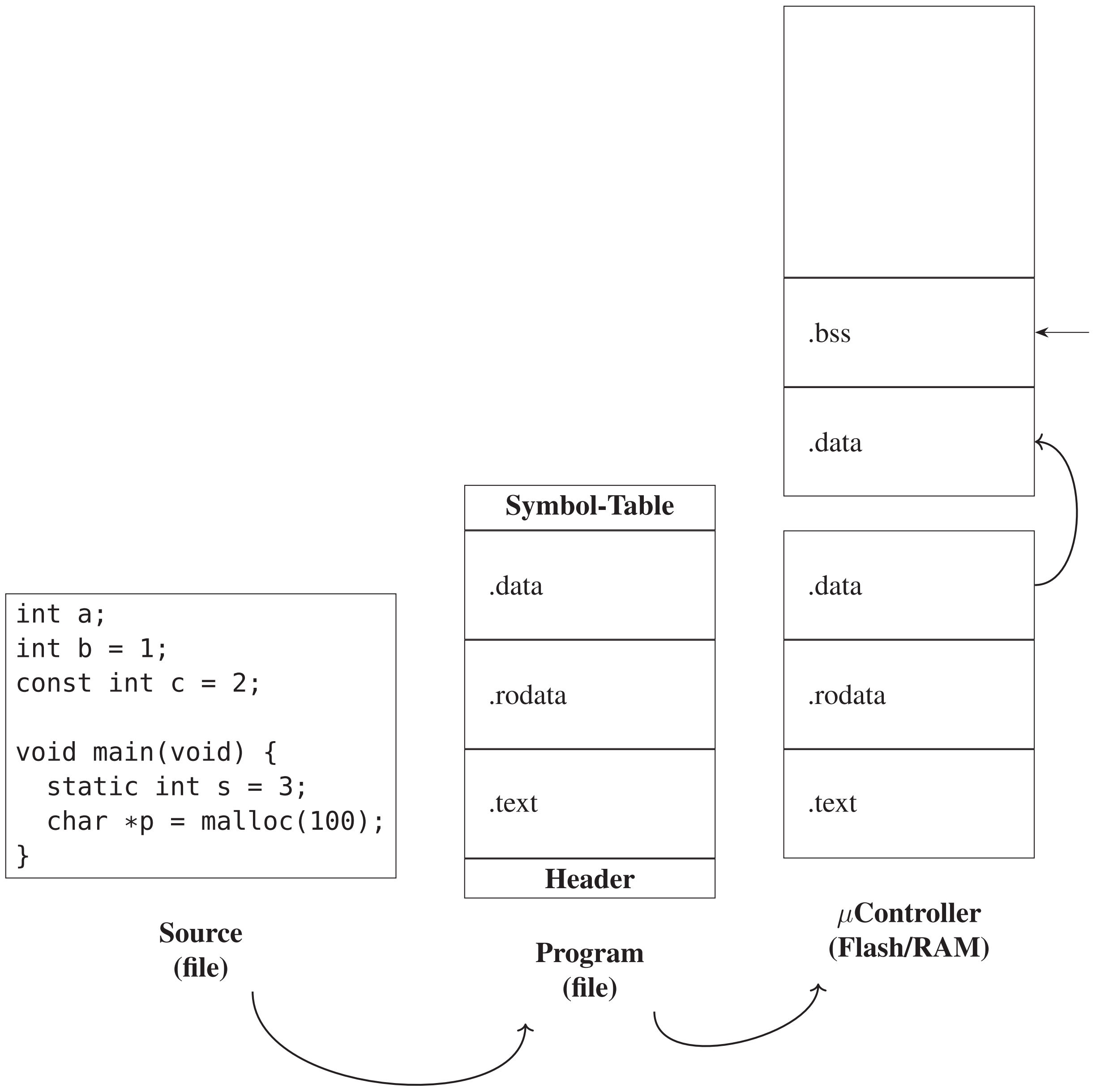
9 Points

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

2) Assign the program code of the `main` function and the variables occurring in the source code and their values to the memory sections in the program file. (2 Points)

3) Assign the program code of the `main` function and all variables **a, b, c, s, p** occurring in the source code and their values to the memory sections shown in Flash/RAM. (3 Points)

4) Label the four arrows in the figure that indicate the steps for executing a program on a μController. (2 Points)



```
int a;
int b = 1;
const int c = 2;

void main(void) {
  static int s = 3;
  char *p = malloc(100);
}
```

Problem 5: Programming language C (10 Points)

a) Consider the following code snippet for a ATmega328PB 8-bit AVR Microcontroller. Answer the following questions in bullet points:

1. What is the meaning of **`static`** in line 1?
2. What is the meaning of the keyword **`volatile`** in example code?
3. What two concurrency problems could occur in this example code?
4. In bullet points: What is the root cause of the concurrency problems?

(5 Points)

```
 1 static volatile uint16_t cars;
 2
 3 ISR(INT2_vect) {
 4     cars++;
 5 }
 6
 7 void main(void) {
 8     init(); // includes a sei()
 9     while(1) {
10         waitsec(60);
11         if (cars > 300) {
12             send(cars);
13         }
14         cars = 0;
15     }
16 }
```

print.h:

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

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

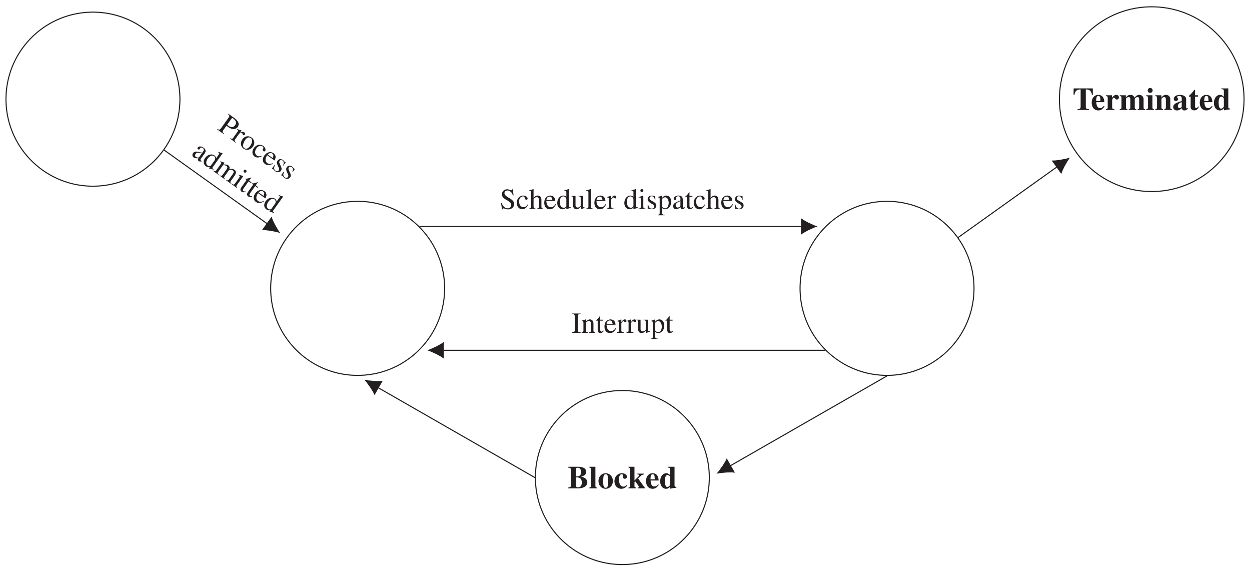
exam.c:

```c
#define WAIT

#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 6: Processes (9 Points)

a) Complete the following graph for the different Linux process states with the corresponding state transitions. Each node corresponds to a process state and each edge corresponds to a state transition. Make sure to label **all** nodes and edges that have not yet been labeled. (3 Points)



Please answer the questions on the next page, too!

b) Describe the process states **Blocked** and **Finished**. (2 Points)

c) Describe two steps that must be carried out for a process switch (**Context Switch**). (2 Points)

d) Name two pieces of information that an operating system (such as Linux) stores in a process control block. (2 Points)