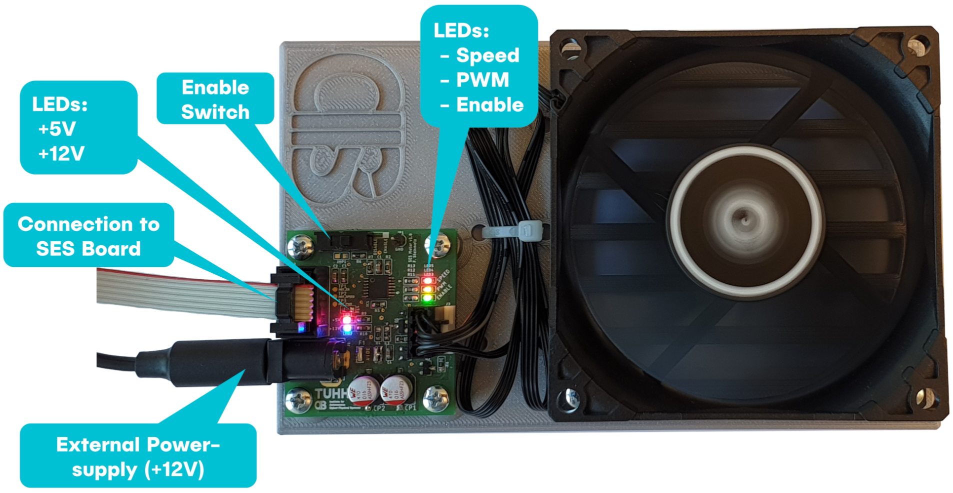
Fan Controller

June 11th, 2025

For this task the SES boards will be extended with a DC cooling fan. The electrical interface of the 4-wire fan provides a PWM control signal, a tacho signal and the two power supply lines. In this lab you implement a program that measures, controls and displays the fan speed.



This is a Graded Exercise Sheet

Evaluation criteria are detailed in *bonus_points.pdf* on StudIP.
The submission deadline is 2025-06-29 23:59 (CET). The interviews will be held during regular lab hours; i. e., on 2025-07-01 (group A) or 2025-07-03 (group B). Both team members must be present.
Commit your solution to *the main branch* of your team's Git repository. Implement the tasks 5.1 to 5.3 in a single project `task_5`. If you do the optional task, use a new project `task_5-4` for it. We expect the following directories/files to be present in your root directory.

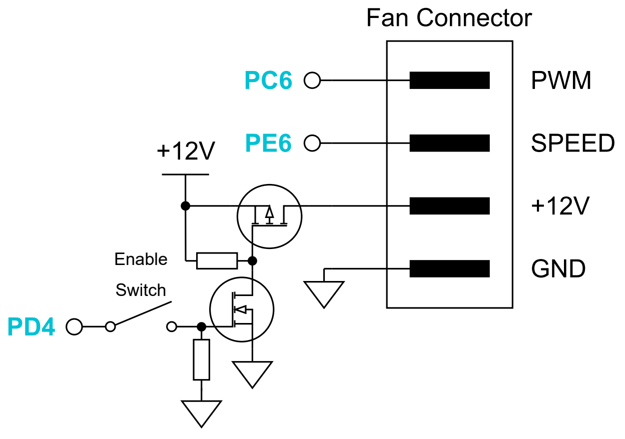
- `lib/ses/` full ses library with all driver files
  - for the PWM: `ses_fan.h` and `ses_fan.c`,
  - for the fan speed: `ses_fanspeed.h` and `ses_fanspeed.c`.
- `task_5` contains source/header files for the main project.
- `task_5-4` contains source/header files for the challenge task (optional).

Ignoring the directory structure will lead to deductions. Ensure to include all **header** and **source** files, but not the compiled binary files.

Task 5.1: PWM Speed Control

Overview:

In this task you control the fan speed via a Pulse Width Modulation (PWM) signal. The fan speed corresponds to the duty cycle of the PWM signal, i.e. higher duty cycle → higher fan speed. The fan's control signal is connected to `PC6` of the microcontroller via the extension board. To control the fan by PWM `PC6` must be configured to use the alternate function `OC3A`: PWM output A of Timer3, controlled by the compare match register `OCR3A`.
The power supply of the fan is en-/disabled by a MOSFET, which is connected to `PD4`. The diagram on the right shows a simplified schematic. Detailed schematics are available on StudIP.



Task Details

Create the header file *ses_fan.h* in the ses library with the following interface:

```c
void fan_init(void);
void fan_enable(void);
void fan_disable(void);
void fan_setDutyCycle(uint8_t dc);
```

Implement the following functionality in *ses_fan.c*:

- Initialize the PWM in `fan_init()`:
  - Configure `DDRD` to control the fan's power supply.
  - Enable Timer3 by clearing bit `PRTIM3` in `PRR1`.
  - Select *Fast PWM, 8-bit* mode via registers `TCCR3A`, `TCCR3B` and `TCCR3C`. Refer to Section 14.8.3 of the datasheet for details.
  - Use the *Compare Output* mode for `OC3A`. Configure the output mode so that a larger value in `OCR3A` (duty cycle) leads to a higher fan speed. Initialize the register with 0.
  - Ensure the PWM period is approximately 1 ms for proper fan operation. Choose a suitable prescaler.
- Set the duty cycle in `fan_setDutyCycle(uint8_t dc)`:
  - Set the value of `OCR3A` based on the parameter `dc`, which ranges from 0 (slowest speed) to 255 (highest speed).
- Implement `fan_enable()` and `fan_disable()` functions to turn the fan's power supply on or off and set the duty cycle to zero.

Test Program

Test your implementation: Create a new project `task_5` with the following functionality:

- Toggle the fan on or off by pressing the push button.
- Use the potentiometer to control the fan speed: Implement a scheduled task that periodically reads the potentiometer and sets the duty cycle.
- Use the scheduler for button debouncing.

Task 5.2 : Frequency Measurement

Overview:

In this task you measure the fan speed using the built-in tacho signal which generates two rising edges per revolution. This digital signal is provided to the microcontroller at pin `PE6` and triggers an external interrupt `INT6` on each rising edge. Use *Timer1* to estimate the fan's frequency in RPM (revolutions per minute).

Task Details:

Create *ses_fanspeed.h* in the ses library with the following interface:

```
void fanspeed_init(void);
uint16_t fanspeed_getRecent(void);
```

Estimating Measurement Range:

Before implementing the fan speed measurement, estimate its expected value range. You can do this by counting the number of edges in a fixed period at minimum and maximum duty cycle. Note that the speed limits might differ due to differing fan models.

- Enable `INT6` (interrupt on a rising edge) and count the number of times the interrupt service routine executes. Toggle the yellow LED in the ISR for visual feedback. Details about *External Interrupts* are given in Chapter 11 of the datasheet.
- Configure `Timer1`:
  - Ensure a time range of at least 1 s.
  - Implement the *Overflow* ISR to display and reset the number of edges during the interval.
- Run the fan at minimum (0% PWM duty cycle) and maximum speed (100% PWM duty cycle)
- Calculate the fan speed in RPM from the number of edges and the timer's range. Remember that one revolution equals two rising edges!

Measuring the Fan Speed:

Choose between the following two variants for speed calculation. Implement the variant you find most suitable and include a short explanation in your code as a comment.

Variant A: Frequency Counting

This variant counts the number of events—i. e., the tacho's rising edges—in a fixed time window.

- Enable `INT6` (interrupt on a rising edge) and count the events.
- Configure `Timer1` as follows:
  - Run the timer in *compare-match mode*.
  - Determine a timer period that ensures multiple detected events at the minimum fan speed but preserves responsiveness.
  - Choose a suitable prescaler.
  - Estimate fan speed by the number of edges within the period.
- Detect a stopped fan in your function; e. g., by detecting a period without a rising edge.

Variant B: Reciprocal Frequency Counting

This variant measures the time between two (or more) events—i. e., the tacho's rising edges.

- Configure `Timer1` as follows:
  - Choose a suitable prescaler to resolve the maximum period between two events (with a safety margin).
  - Enable overflow or compare match interrupt (see below).
- Enable `INT6` (interrupt on a rising edge). In the ISR, estimate fan speed by the elapsed time since the last event.
- Recognize a stopped fan, e.g., to detect malfunction or missing fan power supply. This is indicated by a timeout via the timer's overflow or compare-match interrupt.

Hint

Register `TCNT1` holds the current timer value. Reset the timer by writing the value 0 to `TCNT1`.

Common for both Variants

Implement the following behavior regardless of the chosen variant:

- **Red LED for stopped fan**: Turn on the red LED if (and only if) the fan stops. In this case, `fanspeed_getRecent()` must return 0 RPM. Test your implementation by toggling the `enable` switch on the motor board.
- **Global speed variable**: Store the speed estimate in a *module-private* global variable; its value is returned by `fanspeed_getRecent()`. Beware of potential race conditions.
- **Output**: Use the scheduler to display and refresh the fan speed and the current duty cycle every second.

Task 5.3 : Fan Speed Filtering

Overview

You may have noticed that the results of `fanspeed_getRecent()` fluctuate and are sometimes inaccurate. Passing the individual samples through a *median* filter addresses these problems.

Preparation

This task requires you to implement a median filter, which is independent of the microcontroller. You can implement and test the filter in a standalone C-program without access to the SES board.

Task Details

Add the following function to the fan library:

```c
uint16_t fanspeed_getFiltered(void);
```

The general idea of the median filter is to use an array that stores the last `N` samples. Use this array as a ring buffer, i. e., always replace the oldest measurement in the array when a new fan speed measurement is taken. When `fanspeed_getMedian()` is called, calculate and return the median of all values in the ring buffer.

Median versus Average

Mind that the *Median* is different from the *Average*, or *Mean*, value. Read up on the difference if you are unsure.

- Extend the main program from the previous task to show the filtered fan speed on the display next to the unfiltered one.
- Find a suitable `N` to stabilize the measurement.
- Pay attention to prevent data inconsistency, e. g., if the external interrupt fires during the calculation. However, long atomic blocks should be avoided; wrapping the whole median calculation in a single atomic block is *not* a valid solution.
- Do not use any C standard library functions for the median filter implementation.

Task 5.4 : Closed-Loop Fan Speed Control (Optional)

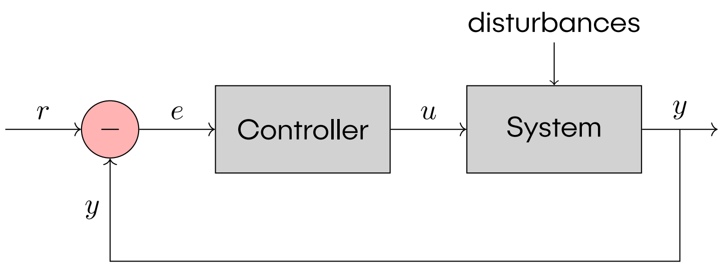
Overview

In Task 5.2, you used the duty cycle to control the fan speed directly in an open-loop system. Due to manufacturing variances, different loads, and installation conditions, the actual fan speed can vary for different fans at the same duty cycle.

In this task, you implement a closed-loop feedback control system to achieve a desired fan speed by using the measured fan speed as feedback. This involves implementing a PID controller on the SES board. After implementing the controller, you write a program that helps you tune the controller's parameters.

PID Controller

A PID controller (as illustrated below) maintains a target fan speed r by using the current fan speed y as feedback. The PID controller calculates a control signal r based on the proportional (P), integral (I), and derivative (D) errors, which is then used to adjust the fan's duty cycle.



PID Algorithm

On a microcontroller, a PID controller with anti-windup (which limits the integral part) can be implemented by periodically executing the following algorithm:

$$\begin{aligned} e &:= r - y \\ e_{\sum} &:= \max(\min(e_{\sum} + e, A_w), -A_w) \\ u &:= K_\mathrm{P} \cdot e + K_\mathrm{I} \cdot e_{\sum} + K_\mathrm{D} \cdot (e_\text{last} - e) \\ e_\text{last} &:= e \end{aligned}$$

- e is the error between the target fan speed and the current fan speed.
- $e_{\sum}$ is the integrated error, limited by A_w.
- u is the output duty cycle.
- e_last is the error from the previous control step.

Task Details

1. **Create a new module** in your SES library and implement the PID Controller.

2. **Tune the parameters** K_P, K_I, K_D, and A_w as detailed below.

Parameter Tuning

Finding suitable parameters can be challenging. **Do not tune them by repeatedly flashing new programs to the microcontroller**. Instead, write a program to modify the parameters dynamically at runtime.

- Use the pushbutton to switch between two target fan speeds r; e.g., 25% and 75% duty cycle.
- Use the rotary encoder to optimize the parameters K_P, K_I, K_D, and A_w:
  - Use the rotary button to cycle through the parameters.
  - Use the rotary encoder to increase or decrease the selected parameter's value.
- Output the current parameter values and intermediate control steps via the USB serial connection.
- Plot the (measured) fan speed on the display to visualize variations and oscillations. This can be achieved with the display library's feature to set individual pixels.
- Use the tuned parameters as defaults in your submission.

How to Tune a PID Controller

Many guides on how to tune a PID controller exist[a]. We suggest to

1. start with $K_\mathrm{P} = K_\mathrm{I} = K_\mathrm{D} = 0$ and twice the maximum fan speed for A_w.
2. Tune the gain parameters.
   - Increase K_P until oscillations occur, then decrease slightly.
   - Adjust K_I (and possibly A_w) to reduce remaining steady-state error.
   - Increase K_D to improve the response time without causing instability.
3. Repeat step two until you are satisfied.

[a] https://pidexplained.com/how-to-tune-a-pid-controller/

Consider the following **hints**:

- Choose the control steps' interval so that a new speed measurement is available in every step.
- Limit the range of the parameters to $0 \leq K_\mathrm{P}, K_\mathrm{I}, K_\mathrm{D} \leq 2 \cdot RPM^{-1}$ and use a resolution of $1/100$ during tuning.
- The template uploaded in StudIP contains a button library with full support for the rotary encoder.
- **Do not use floating point operations.** Rely on fixed point integer calculations. Ensure that intermediate results are cast correctly, and prevent overflows.