Motor Controller

June 23rd, 2020

For this exercise, the SES boards will be equipped with a DC motor. Your task is to implement a motor controller that regulates and measures speed of the motor.

This is a graded exercise. For further information regarding the evaluation criteria read the document *gradedExercises.pdf* carefully. The submission deadline is 2020-07-02 17:00 (CEST). The interviews will be held on Tuesday after submission, i.e., 2020-07-07.

Your solution has to be committed to *the master branch* of your team's Git repository by the specified time. You can implement the tasks 5.1 to 5.3 in a single project `task_5`. If you choose to do the challenge, use a new project `task_5-c` for it. We expect the following directories/files to be present in your root directory.

- `lib/ses/` (contains your current ses library source/header files, including the new drivers `ses_pwm.h`, `ses_pwm.c`, `ses_motorFrequency.h`, and `ses_motorFrequency.c`).
- `task_5` (containing source/header files for the main project)
- `task_5-c` (containing source/header files for the challenge task)

Ignoring this directory structure will lead to point deductions. Make sure that all **header** and **source** files necessary to build and link your solution are present (you must **not** submit the compiled libraries!).

Keep the structure of those directories flat, i.e., do not use subdirectories there!

Task 5.1 : PWM Speed Control

The electric current delivered to the motor can be controlled by a transistor whose base is connected to `PORTG5`. This pin is also the OC0B pin, which can be controlled by the Timer0 hardware, e.g., on compare match. Thus, a PWM signal with a fixed frequency but adjustable duty cycle can be used to control the motor speed. In the SES library, create a file *ses_pwm.h* with the following interface

```
void pwm_init(void);
void pwm_setDutyCycle(uint8_t dutyCycle);
```

Now create the file *ses_pwm.c* and implement the following

- Write a 0 to bit PRTIM0 in PRR0 to enable Timer0.
- Read Section 17.7.3 of the datasheet to understand the functionality of the fast PWM mode and select the respective mode in TCCR0A and TCCR0B.
- Disable the prescaler so that the timer is directly driven from the processor clock. Think about why a high prescaler is unsuitable.
- Configure the timer registers to set the `OC0B` pin when the counter reaches the value of `OCR0B`.
- Implement the function `pwm_setDutyCycle()` to set the value of `OCR0B`.

Test your code by creating a new project `task_5`.

- At initialization, the motor shall be stopped.
- After pressing the joystick button, start the motor by configuring a `OCR0B` value of 170.
- A further press to the joystick button shall stop the motor again.

Unfortunately, the low webcam frame rate in the remote lab environment is not suitable to verify subtle differences in motor speed. However, you should be able to see if a motor is turning or not.

Task 5.2 : Frequency Measurement

The DC motor consits of three motor windings and a commutator. Every time the commutator passes a winding, there is a sharp drop in current. In our motor, this leads to six current drops per revolution. For details of this measurement approach see the footnotes[1]. The circuit we provide filters the current signal and creates a rising edge for every current drop. This digital signal is connected to the ATmega128 at the `PORTD0` pin. The rising edge on this pin can be used as external interrupt `INT0`.

In the SES library create a file *ses_motorFrequency.h* to provide an interface to measure the revolutions of the motor in Hertz with the following functions.

```
void motorFrequency_init();
uint16_t motorFrequency_getRecent();
uint16_t motorFrequency_getMedian();
```

- Similar to the button interrupt pins, a signal edge at `INT0` can generate an interrupt. Read Chapter 16 and configure an interrupt for every rising edge and toggle the yellow LED.
- Use Timer5 for calculating the frequency.
  - Implement an appropriate timer configuration that fulfills the requirements for the following functionality. Choose a correct prescaler to measure frequencies down to 10 Hz.
  - Use the interrupt service routine of `INT0` to measure the time required for one revolution. Remember that one revolution produces six rising edges.
  - Implement a functionality that allows to recognize a stopped motor, e.g. by using the CTC mode and the timer interrupt to notice a period without a rising edge. While a stopped motor is recognized, the green LED shall light up, and the get functions shall return a frequency of 0 Hz.
- Implement `motorFrequency_getRecent()` to return the most recent measurement in Hertz.
- In the `task_5` project, use the scheduler to display the motor frequency in revolutions per minute (rpm) on the LCD with an update rate of one second.

The current value of the timer can be read out from `TCNT5`. The timer can be reset by `TCNT5 = 0`.

Task 5.3 : Median Calculation

As you might notice, the results of `motorFrequency_getRecent()` are quite variable and many erroneous measurements occur. Therefore, the measurements should be filtered with a median filter. Implement the function `motorFrequency_getMedian()` that calculates the median of the last `N` interval measurements.

In the interrupt service routine, store the samples in a suitable data structure that can take up to `N` samples. When `motorFrequency_getMedian()` is called, calculate the median over these measurements and return the inverse in Hertz. Show the result in rpm on the LCD together with the result of `motorFrequency_getRecent()`.

[1] For details see https://www.precisionmicrodrives.com/tech-blog/2011/06/08/using-dc-motor-commutation-spikes-to-measure-motor-speed-rpm

- Find a suitable `N` to allow for a stable but responsive measurement.
- Pay attention to prevent data inconsitencies if the external interrupt fires during the calculation. However, avoid long atomic blocks, so wrapping the whole median calculation in a single atomic block has to be avoided.

Task 5.4 : Challenge

In Task 5.2 you have used a fixed duty cycle to control the motor speed. However, due to production variances and different motor loads, this does not lead to a constant, reproducible motor speed. Implement a PID controller to maintain a given motor frequency f_{target} once the joystick button is pressed, while measuring the current motor frequency f_{current} concurrently. As a first step, plot the trace of the frequency on the LCD to better see variations and oscillations. For this purpose, the LCD library allows you to set single pixels of the LCD.

On a microcontroller a PID controller with anti-windup can be implemented by executing the following algorithm given as pseudo code in regular intervals.

$$
\begin{aligned}
e &:= f_{\text{target}} - f_{\text{current}} \\
e_\Sigma &:= \max(\min(e_\Sigma + e, A_w), -A_w) \\
u &:= K_p \cdot e + K_I \cdot e_\Sigma + K_D \cdot (e_{\text{last}} - e) \\
e_{\text{last}} &:= e
\end{aligned}
$$

In these equations, e is the error of the current frequency from the target frequency, e_Σ is the integrated error, A_w limits the integration (anti wind-up) and u is the duty cycle that must be applied to the motor.

Find appropriate values for K_p, K_I, K_D and A_w and initialize the controller properly.

- There are many guides on the internet how to tune a PID controller from scratch. One is linked in the footnotes[2].
- Do not use floating point operations. Thus, you have to scale the calculations.
- Use the UART for debugging the development of the variables.

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