**DTEK0068**

**Embedded Microprocessor Systems**

# Optical Tachometer

Project Report
18.12.2020

Sandra Ekholm
svekho@utu.fi
518981

Nea Kontturi
nckont@utu.fi
520285

Amy Nymalm
aanyma@utu.fi
518716

# 1 Assignment Description and Analysis

Functional requirements for optical tachometer include using visible light to measure RPM of small DC motor and a propellor. For measuring visible light, we decided to use LDR which we connected to ADC channel AN8. ADC would compare the received value from LDR to its defined voltage reference. The DC motor shall implement an adjustable motor RPM control. The control was settled with a potentiometer which was connected to ADC channel AN14 with different voltage reference. Because ADC is able to receive information from only one channel these two inputs and their voltage references would be handled alternately.

To be able to control the rotating speed of the motor some kind of signal needs to be created. For this we decided to use PWM signal and by upgrading its duty cycle, the rotation speed will change. To create continuous PWM signal, we chose TCB to take care of signal control in 8-bit PWM mode. Period time was selected to be one second and the duty cycle would be updated based on values measured from potentiometer.

LCD display was decided to visualize the measured and calculated RPM. Functional requirement for updating RPM values was to update it once a second. For updates once a second we discussed RTC would fulfil this criterion satisfyingly. RTC would need to be configured to produce periodic interrupts, and RTC clock and clock cycles would have to be settled to count up to one second.

Now when RTC and ADC were decided to be included as part of a project, we would need two different kinds of interrupts one for each. RTC was configured to produce interrupts once a second, and ADC was configured to produce interrupts every time it produced a result. Because power saving mode was desired, the challenge was to recognize which interrupt was concerned to wake the CPU from sleep. For this, an identifier was created. Based on that, proper actions would be executed.

The actions, that we needed to implement, were LCD display updating, rotations counting from LDR and new rotation speed setting based on potentiometer. Another challenge was to specify which ADC interrupt values would concern LDR measurements and which potentiometer measurements. This was solved with another identifier. We decided LDR was a component which would need more attention than potentiometer, so every 100th ADC value would be measured from potentiometer and others from LDR. When this counter reached 99, the ADC channel would be switched to potentiometer. When counter reached 100 and result was ready from ADC, the PWM duty cycle was changed to value based on measurement. We needed ADC resolution to be as precise as possible but the duty cycle register in TCB would accept only 8 bits. This problem was solved with right shift operation of C language. When the duty cycle was chanced correctly, the ADC channel was updated back to measure LDR.

We encountered many challenges with LDR measuring. After arranging an ambient lightning calibration and another identifier to detect propellor spinning more precisely, so that it wouldn't measure same propellor state twice, these challenges were sort out satisfyingly. LCD updating didn't produce any challenges worth mentioning.

# 2 Solution

## 2.1 Repository Link to GitHub

To begin with the solution of the project, this is the link to the repository in GitHub:
https://github.com/AmyNyma/Optical_Tachometer/tree/master/optical_tachometer_FINAL.X

## 2.2 Variables

Global variables in file main.c include: `adcValue`, `isPropOn`, `lcdUpdate`, `voltThreshold` and `potentRead`. Those, that are declared as `volatile`, are used both in `main()` and in `ISR`:s, which indicates that compiler should not avoid "unnecessary" fetches from memory. The variables' values may change at any time without any action of the code compiler finds nearby. The variable `voltThreshold` is also used in more than one function, but its value is constant so it doesn't need continuous fetches from memory. Data types are all unsigned integers with different widths. Unsigned integer is used because it is not possible for some variables to have negative values, and due to coherence of the code they are all defined as unsigned. For developers, it was also possible to use unsigned values for those variables that didn't make difference whether values were negative or positive. The operations between unsigned integer types are more stable compared to different types of data variables. Variables' widths were decided observing principle in which data memory shall be allocated as little as possible.

Local variables' widths in `main()` function were designed based on same principle as with global variables. Variable `rotations` is considered as `volatile` to ensure the compiler has correct value all the time. The value changes often and in different parts of code, which may not be code compiler finds nearby. It would be really bad situation if the compiler does not fetch new values often enough. Other local variables in `main()` don't need to be `volatile` because they appear only in overwritten operations and they are used immediately after updating. Variables are introduced local because they appear only in the specified function. Other variables declared in other functions are also local variables due to same reason. Principles defining their data types and widths are same as introduced before.

## 2.3 Initialization Functions

Initializations needed for this project are RTC, ADC, TCB, LCD and USART. USART is configured to be an interface between the microcontroller and user. Through USART the program sends information for the user when the program starts via putty or computer terminal. The ambient lightning for the use of LDR is calibrated in the beginning of the program, and the user will be informed about the start of the calibration and when the calibration is ready. Also the calculated value will be represented with the help of USART. In function `ldr_threshold_calibrate()` LDR measures the light and ADC converts the results 10000 times from which every 3333th result will be stored in an array. Value of `voltThreshold` will be the average result from those three values.

RTC is initialized into periodic interrupt mode. LCD shall be updated once a minute which means RTC's clock needs to be configured into proper mode for this. We decided to use External Clock Oscillator and this means we needed 32768 clock cycles to reach interrupt once a minute.

At the same time with initializing ADC, we initialized correct input pins for LDR and potentiometer. Port E pin 0 (AN8) was selected for LDR and Port F pin 4 (AN14) for potentiometer. Reference voltage for LDR was set to 1,5 V which can be accessed from internal reference voltage. Potentiometer's reference voltage is VDD, and it was set when it was time to measure potentiometer. To get precise ADC values, resolution was set to 10 bits. Freerun-mode was selected because with this mode the next conversion for ADC would start automatically.

TCB and its output pin to DC motor was initialized simultaneously. Port A and pin 2 was configured as output. TCB was configured in 8-bit PWM mode because we made a conclusion that it would fulfil our needs best, because we didn't need that precise dual-slope PWM signal or two different PWM signals, which both would have been provided by TCA. When initializing TCB we set the period of one second to CCMPL register, and the first duty cycle was initialized as 50% in CCMPH register.

LCD initialization is located in other source file `lcd.c`. Function `lcd_init()` sets all the pins LCD uses as outputs: from Port B pins 3-5 and from Port D pins 0-7. LCD backlight is also enabled at the same time. Vports are used through entire code if there is a possibility because configuring single pins, they provide atomic operations and with many pins they provide faster operations.

## 2.4 Update Functions

Code has two different update functions which are located in separate files: `lcd_update(uint16_t rpm)` in `lcd.c` and `spin_update(uint8_t userVoltage)` in `spin.c`. `Spin_update()` receives the potentiometer value measured by ADC. This function disables the TCB peripheral to ensure proper update process for the duty cycle. The CCMPH register will be updated with the same value received from `main()`. After updating the duty cycle, the periphperal is enabled again.

In `lcd_update()`, before actually updating LCD the digits of the received rpm value are calculated. This is done because we decided to create a string in which the digits would be placed as chars. The digits are placed into the string in a for loop. After this, the LCD display is instructed to clear the display from previous rpm value, and to set the cursor back at the left home position. And finally, the value of rpm is sent to display looping the created string digit by digit. Display is instructed to set a "space" and a string "RPM" after the value.

## 2.5 Main

First, all local and global variables are initialized to 0. Then all the initialization functions are called. Before calling the lightning calibration function the ADC conversion is started because calibration needs LDR conversion values. Global interrupts are enabled before starting the superloop.

CPU is set in idle sleep mode and it is instructed to enter sleep in the beginning of superloop. Only interrupts will wake it to process other code in the superloop. There are three possible interrupts to wake CPU from sleep, and each interrupt has its own indicator: variable `lcdUpdate`. Operations based on each interrupt are entered in `if` and `else if` conditions.

When interrupt was from RTC the LCD shall be updated. In superloop under this condition in `if-else` structure, the RPM is calculated and function `lcd_update` is called with calculated value. Instructions are surrounded by `cli()` and `sei()` functions. This is to ensure proper execution of LCD updating and to prevent unstable states and data corruptions.

When interrupts was from ADC, the same indicator `lcdUpdate` conducts which operations shall be performed. If the indicator is 2 the ADC result is received from LDR. This section has more `if-else` structures because ADC result might tell different things from LDR. Propellor might have been in front of the LDR and the rotation counter shall be updated, or the `potentRead` counter might indicate that the ADC channel shall be switched. If these conditions are not met only the `potentRead` counter is updated. When one of the cases is entered the operations inside are again surrounded by `cli()` and `sei()`. This is to ensure atomic operations and proper ADC channel switch, and to prevent corruption of data.

If interrupt was from potentiometer ADC conversion, interrupts must be disabled for the duration of spin updating and ADC channel switching for same reasons as described above. There are delays when switching ADC channel and its reference voltage. These are necessary for ADC but hold interrupts away which might be the reason for program not processing data correctly. Delays shall not be used in this case, because interrupts must be released as fast as possible to ensure correct execution of program. Also when updating the LCD, the operation is long and includes also necessary delays for LCD.  This is not good situation for code executing and interrupts.

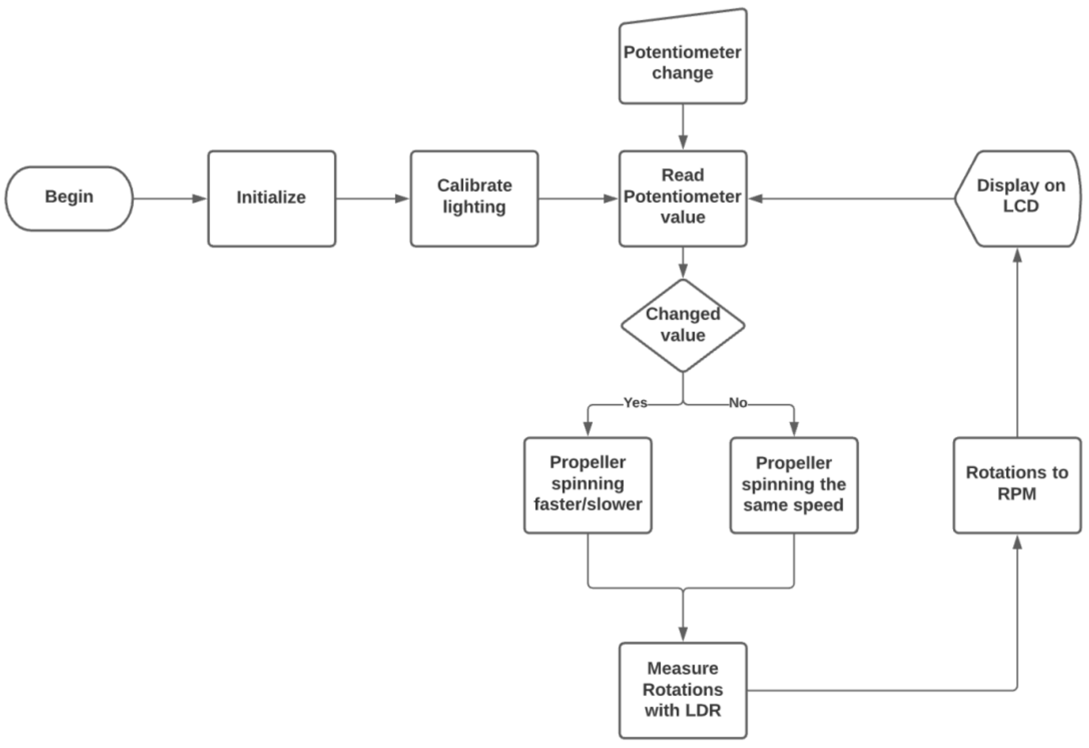
Certain variables must be updated simultaneously without interrupts to prevent conflicting data situations. ADC and other updates shall also be executed without interrupts because peripherals might also end up in conflicts. `Cli()` and `sei()` functions cannot be set to surround entire code block after sleep because it bans the interrupts totally. That’s why these functions are set inside each if-condition. Interrupts may occur when moving forward from if-condition to other and this might also end up in data corruption. Some code, which would be necessary to execute, might not be executed and some other parts might be executed instead.

## 2.6 Interrupts

Interrupts are designed to be as short as possible. In RTC interrupt and in ADC interrupt both interruptflags are cleared. Indicator for different operations are initialized based on the interrupt type.

ADC interrupt has more operations in it compared to RTC interrupt. That is because ADC interrupt must identify whether interrupt was from potentiometer or LDR. In case with LDR, proplleor state must also be recognized and set.

## 2.7 Flowchart



## 3 Testing

Testing our code was mainly done through observing how the components connected to the ATMEGA reacted to light change in the LDR and turning on the potentiometer. Putty was also used so we were able to easily observe the different variable values and see if they were correct.

The 7-segment display was tested by comparing the display output to putty output. When the same value was printed to putty and the 7-segment display, we concluded our code worked. We tested the values from the LDR and the potentiometer the same way, by changing the input (lighting on LDR and turning the potentiometer) and observing what was printed to putty. We never knew if the values printed to putty were exactly correct, but we were satisfied when they seemed logical. The printing to the LCD screen was easy to test once the LCD worked, which was not that simple. If the value seemed right with a few spins in front of the LDR we were satisfied.

We tested our way to a suitable `MIN_VOLT_DIFF` value for our `voltThreshold` variable by changing the value and checking how the `RPM` value changed on LDR input. We found a suitable value, that was small enough to detect a propellor above the LDR, but large enough to discard any small changes in the lighting. Our largest problems were with the motor. Our equipment was apparently not working so well, so we deemed the code working, when our teacher made a small change in the code and said it worked for him. (Thank you Jani)

In short, we tested our code with different variations on input, small changes in code and in variable sizes, and when the components did what we expected them to do on multiple tries, we concluded the code worked. It was not easy though with a lot of not working components.

# 4 User Guide

The device has 7 parts that the user should be aware of:
-Potentiometer x2 (in the guide P1 and P2)
-LDR
-LCD display
-serial to USB cable
-micro-USB to USB cable
-Motor with propellor

## 4.1 Preparation

The micro-USB cable has to be connected to a current source (computer) for the device to work. The serial to USB cable is optional and does not affect the functionality of the device. Make sure there is a light source (preferably a single led) in front of the LDR, but far enough away for the propellor to fit in between. Other light sources are ideally turned off.

## 4.2 Use

While the device is started the propellor **cannot** be in front of the LDR. The device needs a few seconds for light calibration, and it will not work if the propellor is in front of the light source during that time. After the calibration, the program has to be started again if the light source is moved or changed.

When calibration is complete, the program is good to go! P1 controls the speed of the motor. The motor is in turn used to rotate the propellor between the light source and the LDR. The rotation speed is measured by the LDR and is shown on the LCD-display in RPMs. The other potentiometer (P2) is used to adjust the brightness on the LCD. If the LCD does not show any RPMs, the issue might lie in the brightness.