

Ministerul Educației al Republicii Moldova

Universitatea Tehnică a Moldovei

# RAPORT

la Programarea aplicațiilor încorporate și independente de platformă

Lucrare de laborator Nr. 5

Tema: : Timers and Interrupts

A efectuat st. gr. FAF-141:

Oxana Dunav

A verificat:

Andrei Bragarenco

Chișinău 2016

## Topic

Timers. Interrupts

## Purpose

- Implement a task runner (scheduler)
- Implement descriptor for task
- Use timers for controlling tasks in time.

## Task

To develop a small application that will turn on and off a set of LEDs, one at a time.

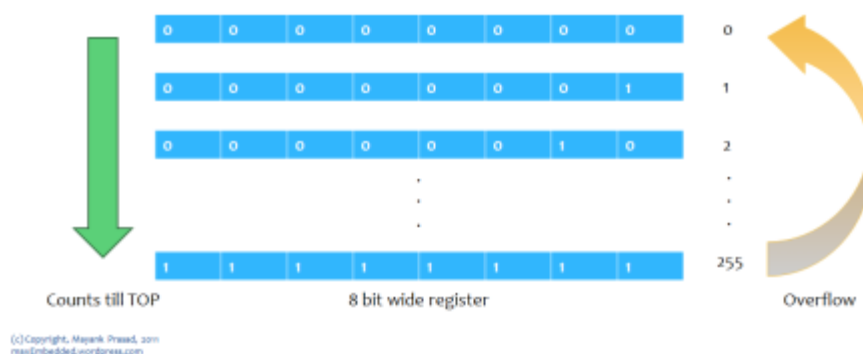
## Domain

### → Timers

Timers are used everywhere. Without timers, you would end up nowhere! The range of timers vary from a few microseconds (like the ticks of a processor) to many hours, and AVR is suitable for the whole range! AVR boasts of having a very accurate timer, accurate to the resolution of microseconds! This feature makes them suitable for timer applications. Let's see how.

### → Timers as registers

So basically, a timer is a register! But not a normal one. The value of this register increases/decreases automatically. In AVR, timers are of two types: 8-bit and 16-bit timers. In an 8-bit timer, the register used is 8-bit wide whereas in 16-bit timer, the register width is of 16 bits. This means that the 8-bit timer is capable of counting  $2^8=256$  steps from 0 to 255 as demonstrated below.



8 bit Counter

Similarly a 16 bit timer is capable of counting  $2^{16}=65536$  steps from 0 to 65535. Due to this feature, timers are also known as counters. Now what happens once they reach their MAX? Does the program stop executing? Well, the answer is quite simple. It returns to its initial value of zero. We say that the timer/counter **overflows**.

In ATMEGA32, we have three different kinds of timers:

- TIMER0 – 8-bit timer
- TIMER1 – 16-bit timer
- TIMER2 – 8-bit timer

The best part is that the timer is totally independent of the CPU. Thus, it runs parallel to the CPU and there is no CPU's intervention, which makes the timer quite accurate.

Apart from normal operation, these three timers can be either operated in normal mode, CTC mode or PWM mode.

#### ➔ The Prescaler

Assuming  $F_{CPU} = 4 \text{ MHz}$  and a 16-bit timer ( $MAX = 65535$ ), and substituting in the above formula, we can get a maximum delay of 16.384 ms. Now what if we need a greater delay, say 20 ms? We are stuck?!

Well hopefully, there lies a solution to this. Suppose if we decrease the  $F_{CPU}$  from 4 MHz to 0.5 MHz (i.e. 500 kHz), then the clock time period increases to  $1/500k = 0.002 \text{ ms}$ . *Now* if we substitute **Required Delay = 20 ms** and **Clock Time Period = 0.002 ms**, we get **Timer Count = 9999**. As we can see, this can easily be achieved using a 16-bit timer. At this frequency, a maximum delay of 131.072 ms can be achieved.

Now, the question is how do we actually reduce the frequency? This technique of frequency division is called prescaling. We do not reduce the actual  $F_{CPU}$ . The actual  $F_{CPU}$  remains the same (at 4 MHz in this case). So basically, we derive a frequency from it to run the timer. Thus, while doing so, we divide the frequency and use it. There is a provision to do so in AVR by setting some bits which we will discuss later.

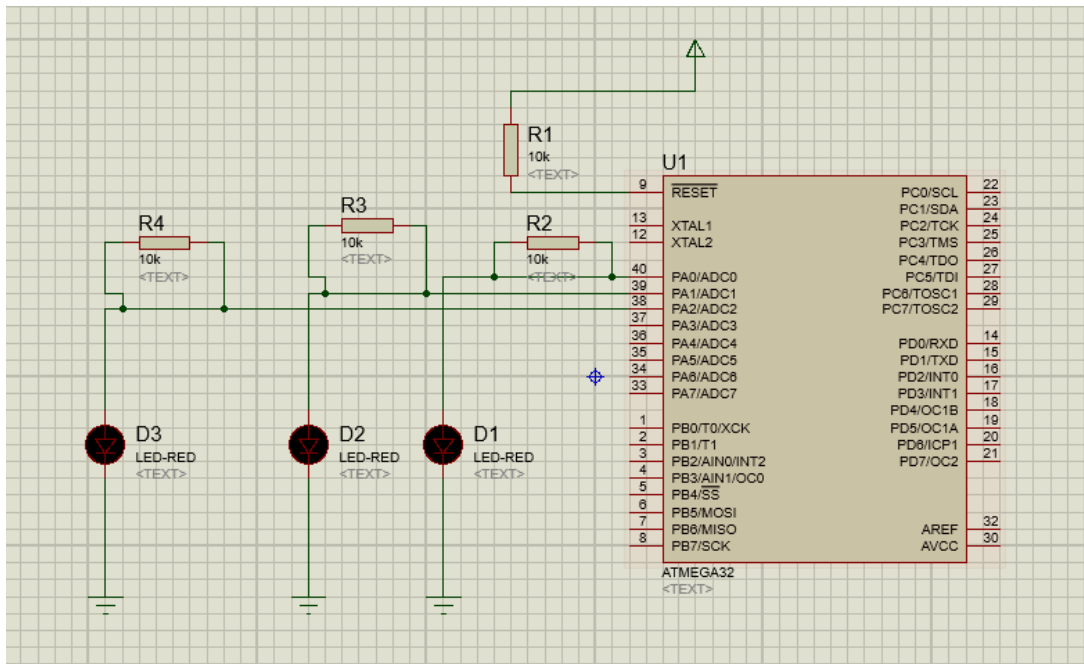
But don't think that you can use prescaler freely. It comes at a cost. There is a trade-off between resolution and duration. As you must have seen above, the overall duration of measurement has increased from a mere 16.384 ms to 131.072 ms. So has the resolution. The resolution has also increased from 0.00025 ms to 0.002 ms (technically the resolution has actually decreased). This means each tick will take 0.002 ms. So, what's the problem with this? The problem is that the accuracy has decreased. Earlier, you were able to measure duration like 0.1125 ms accurately

( $0.1125/0.00025 = 450$ ), but now you cannot ( $0.1125/0.002 = 56.25$ ). The new timer can measure 0.112 ms and then 0.114 ms. No other value in between.

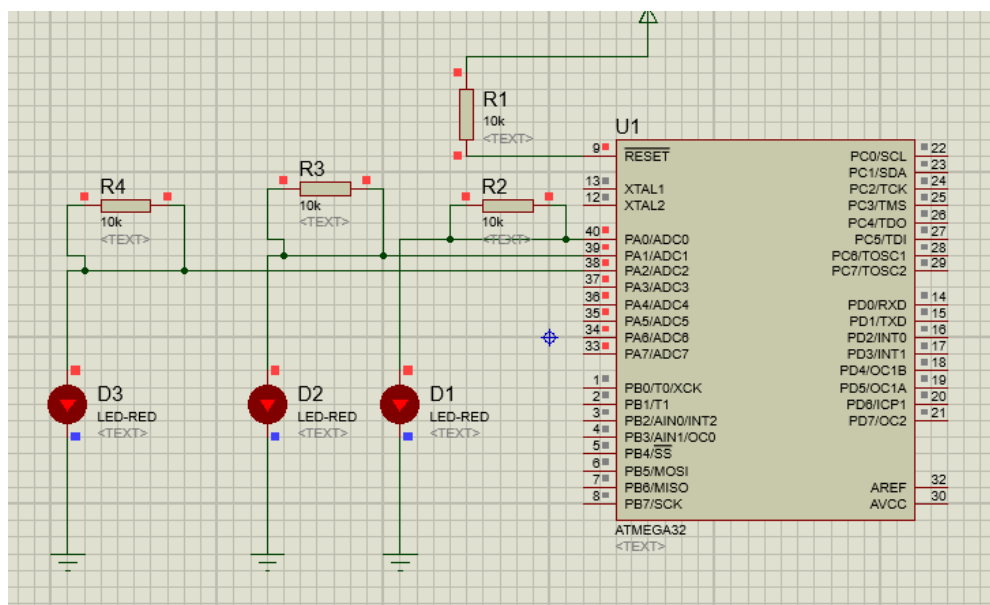
## Solution

### Schematics

For our laboratory work we need ATmega32 MCU and 3 leds that will run according to the timer.



### Simulation Result



## Conclusion

During this laboratory work I learned how to work with Timers in the microcontroller ATMEGA32 and interrupts and also I've learned how to prescale the processor's speed.

This gives us possibility to have different tasks running that are controlled using the timer.

## Appendix

### main.c

```
#include <avr/io.h>
#include <avr/interrupt.h>

// *****
// Interrupt Routines
// *****

uint32_t counter = 0;

// timer0 overflow
ISR(TIMER0_OVF_vect) {
    // XOR PORTA with 0x01 to toggle the second bit up

    toggle_led(counter);
    counter++;
    if(counter >= 3) {
        counter = 0;
    }
}

int main( void ) {
    // Configure PORTA as output
    DDRA = 0xFF;
    PORTA = 0xFF;
    // enable timer overflow interrupt for both Timer0 and Timer1
    TIMSK=(1<<TOIE0) | (1<<TOIE1);
    // set timer0 counter initial value to 0
    TCNT0=0x00;

    TCCR1B |= (1 << CS01);
    // enable interrupts
    sei();
    while(1) {
    }
}
```

### led.h

```
#ifndef SRC_LED_H_
#define SRC_LED_H_
#include <avr/io.h>
```

```

#include "stdint.h"

void init_led();
void turn_on(uint32_t pin);
void turn_off(uint32_t pin);
void toggle_led(uint32_t pin);

#endif /* SRC_LED_H_ */

```

## led.c

```

#include "led.h"
#include <stdint.h>
#define MAX 5

void init_led() {
    DDRA |= 0xFF;
}

void turn_on(uint32_t pin) {
    PORTA |= pin;
}

void turn_off(uint32_t pin) {
    PORTA &= pin;
}

void toggle_led(uint32_t pin) {
    PORTA ^= (1 << pin);
}

```