Ministerul Educației al Republicii Moldova

Universitatea Tehnică a Moldovei

RAPORT

la Programarea aplicațiilor incorporate 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

Chişinău 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 applicationa that will turn on and off a set of LEDS, one at a time.

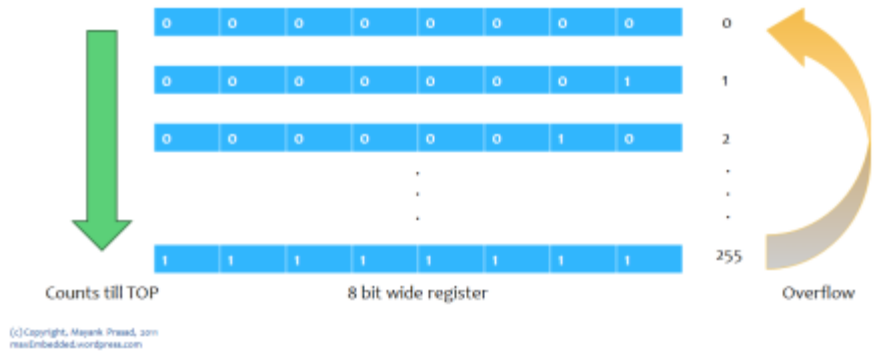
Domain

➔ 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.

➔ TCNT0 Register

The **Timer/Counter Register** – TCNT0 is as follows:

Bit	7	6	5	4	3	2	1	0	
	TCNT0[7:0]								TCNT0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TCNT0 Register

This is where the uint 8-bit counter of the timer resides. The value of the counter is stored here and increases/decreases automatically. Data can be both read/written from this register.

➔ TCCR0 Register

The **Timer/Counter Control Register** – TCCR0 is as follows:

Bit	7	6	5	4	3	2	1	0	
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
Read/Write	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TCCR0 Register

Right now, we will concentrate on the highlighted bits. The other bits will be discussed as and when necessary. By selecting these three **Clock Select Bits**, **CS02:00**, we set the timer up by choosing proper prescaler. The possible combinations are shown below.

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/(No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

➔ TIMSK Register

The **Timer/Counter Interrupt Mask** – TIMSK Register is as follows. It is a common register for all the three timers. For TIMER0, bits 1 and 0 are allotted. Right now, we are interested in the 0th bit **TOIE0**. Setting this bit to '1' enables the TIMER0 overflow interrupt.

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

➔ Enabling Global Interrupts

In the AVRs, there's only one single bit which handles all the interrupts. Thus, to enable it, we need to enable the global interrupts. This is done by calling a function named sei().

➔ The Prescaler

Assuming F_CPU = 4 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 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.

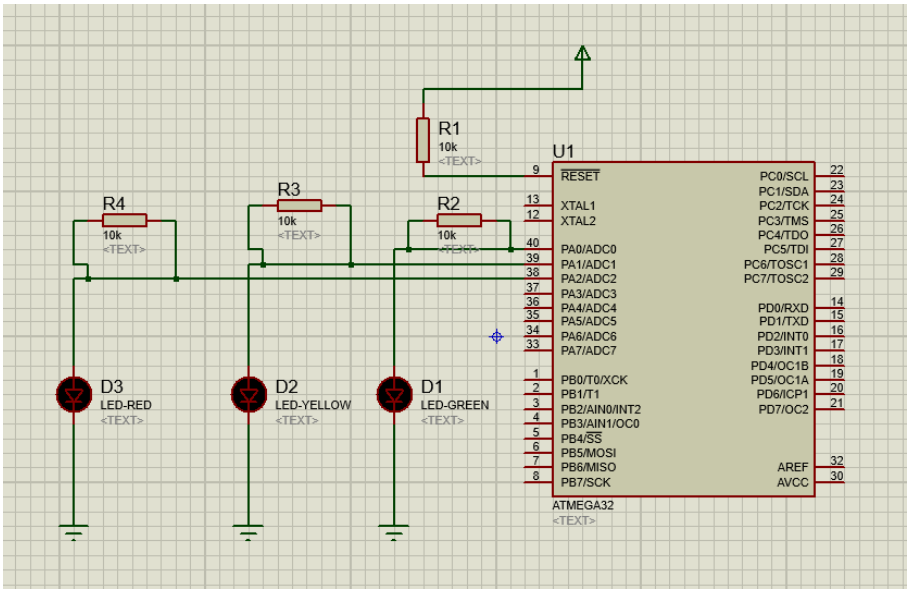
Now, the question is how do we actually reduce the frequency? This technique of frequency division is called prescaling. We do not reduce the actual F_CPU. The actual F_CPU remains the same (at 4 MHz in this case). So basically, we derive a frequency from it to run the timer. Thus, while doing so, we 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

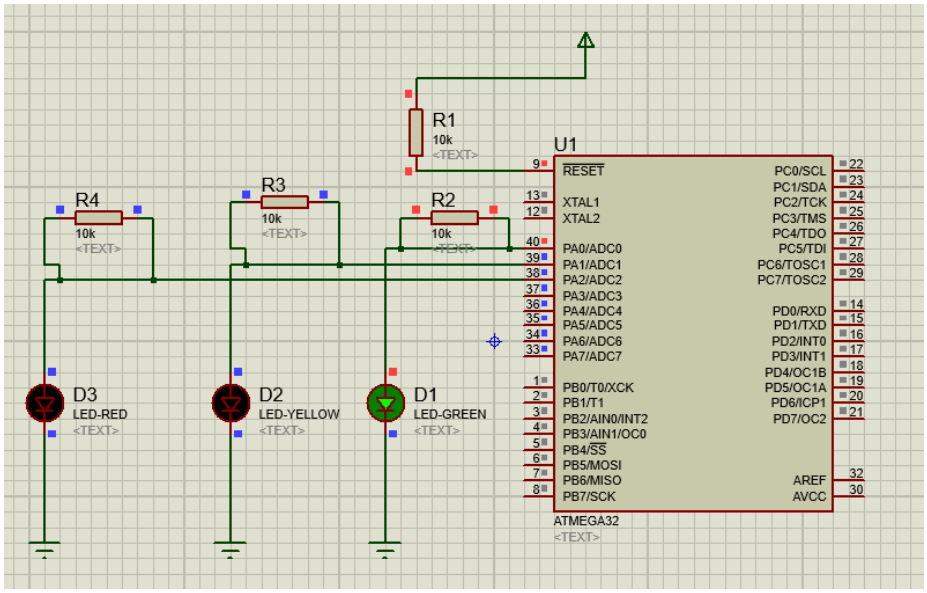
Schematics

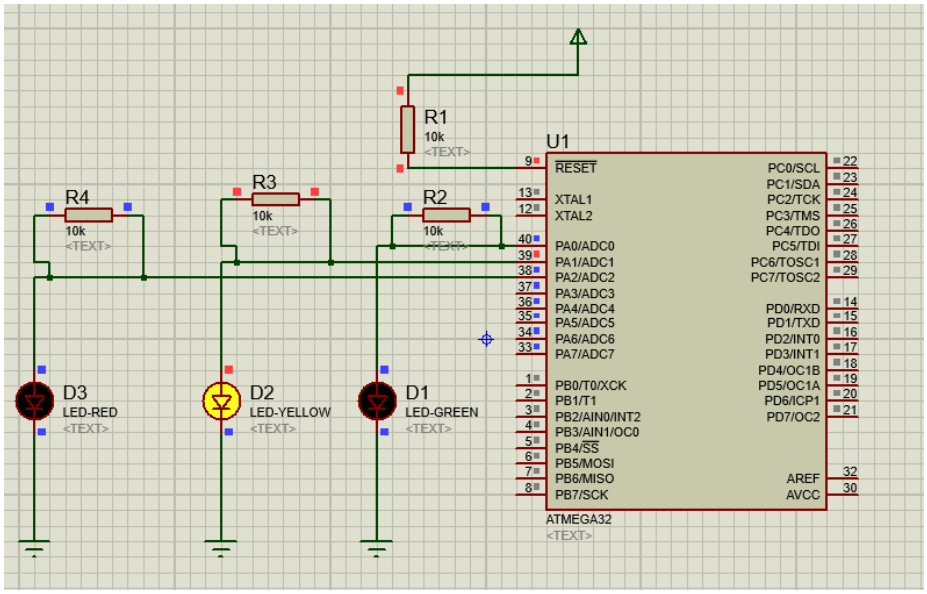
For our laboratory work we need ATMega32 MCU and 3 leds that will run according to the timer.

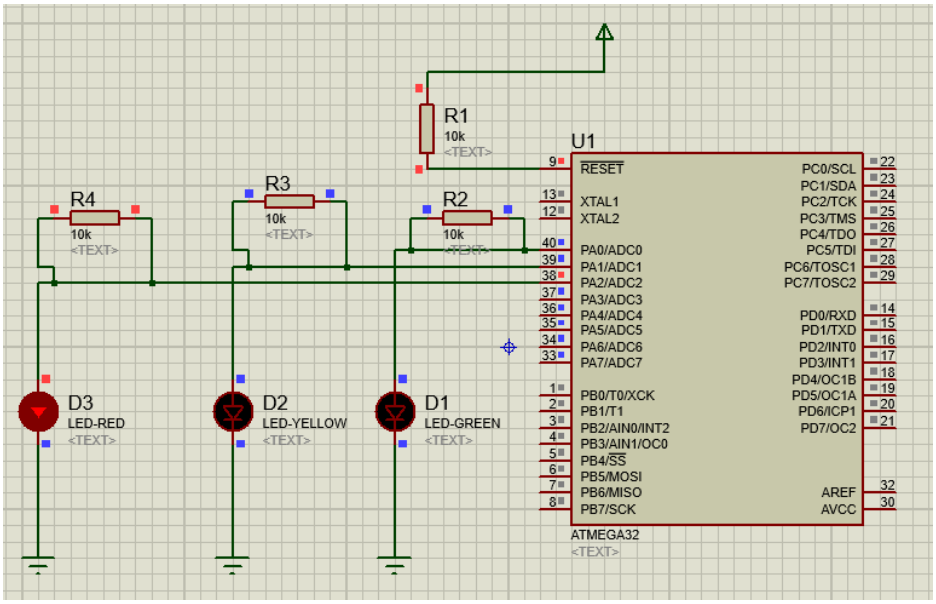


Simulation Result

After some time, the led turn on one by one.


R1
10k
<TEXT>
U1
R4
10k
<TEXT>
R3
10k
<TEXT>
R2
10k
<TEXT>
RESET
XTAL1
XTAL2
PA0/ADC0
PA1/ADC1
PA2/ADC2
PA3/ADC3
PA4/ADC4
PA5/ADC5
PA6/ADC6
PA7/ADC7
PB0/T0/XCK
PB1/T1
PB2/AIN0/INT2
PB3/AIN1/OC0
PB4/SS
PB5/MOSI
PB6/MISO
PB7/SCK
PC0/SCL
PC1/SDA
PC2/TCK
PC3/TMS
PC4/TDO
PC5/TDI
PC6/TOSC1
PC7/TOSC2
PD0/RXD
PD1/TXD
PD2/INT0
PD3/INT1
PD4/OC1B
PD5/OC1A
PD6/ICP1
PD7/OC2
AREF
AVCC
ATMEGA32
<TEXT>
D3
LED-RED
<TEXT>
D2
LED-YELLOW
<TEXT>
D1
LED-GREEN
<TEXT>


R1
10k
<TEXT>
U1
R4
10k
<TEXT>
R3
10k
<TEXT>
R2
10k
<TEXT>
RESET
XTAL1
XTAL2
PA0/ADC0
PA1/ADC1
PA2/ADC2
PA3/ADC3
PA4/ADC4
PA5/ADC5
PA6/ADC6
PA7/ADC7
PB0/T0/XCK
PB1/T1
PB2/AIN0/INT2
PB3/AIN1/OC0
PB4/SS
PB5/MOSI
PB6/MISO
PB7/SCK
PC0/SCL
PC1/SDA
PC2/TCK
PC3/TMS
PC4/TDO
PC5/TDI
PC6/TOSC1
PC7/TOSC2
PD0/RXD
PD1/TXD
PD2/INT0
PD3/INT1
PD4/OC1B
PD5/OC1A
PD6/ICP1
PD7/OC2
AREF
AVCC
ATMEGA32
<TEXT>
D3
LED-RED
<TEXT>
D2
LED-YELLOW
<TEXT>
D1
LED-GREEN
<TEXT>



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>
#include <util/delay.h>


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

uint32_t i = 0;

// TIMER0 overflow interrupt service routine
// called whenever TCNT0 overflows
ISR(TIMER0_OVF_vect) {
    turn_off(0);
       turn_off(1);
       turn_off(2);
       toggle_led(i);
       i++;

       if(i > 2) {
              i = 0;
```

```
	}
}


int main( void ) {
    // Configure PORTA as output
    init_led();

    //Enable Overflow Interrupt Enable for Timer 0
    TIMSK= (1<<TOIE0);
    // set timer0 counter initial value to 0
    TCNT0=0x00;

    // Prescaler = FCPU/1024
    TCCR0 |= (1 << CS02) | (1 << CS00);
    // enable interrupts(set global interrupt flag)
    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 toogle_led(uint32_t pin);

#endif /* SRC_LED_H_ */
```

led.c

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

void init_led() {
	DDRA |= 0xFF;
	PORTA = 0x00;
}

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);
}
```

Flowchart