Ministerul Educației şi Tineretului al Republicii Moldova

Universitatea Tehnică a Moldovei

Departament „Informatica aplicată”

RAPORT

Lucrarea de laborator nr. 5

LA DISCIPLINA”Programarea aplicaţiilor incorporate şi independente de platformă ”

Tema : Timers and Interrupts

A efectuat:

St. gr. FAF 141 A. Dariev

A verificat:

Lect . supp A.Bragrarenco

Chişinău 2017

Topic: Get familiar with timers and interrupts in microcontroller Atmega 32

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

Short theory:

Every electronic component works on a time base. This time base helps to keep all the work synchronized. Without a time base, you would have no idea as to *when* to do a particular thing. Thus, timers is an important concept in the field of electronics. You can generate a time base using a timer circuit, using a microcontroller, etc. Since all the microcontrollers work at some predefined clock frequency, they all have a provision to set up timers.

AVR boasts of having a timer which is very accurate, precise and reliable. It offers loads of features in it, thus making it a vast topic. In this tutorial, we will discuss the basic concepts of AVR Timers. We will not be dealing with any code in this tutorial, just the concepts. The procedure of generating timers and their codes will be discussed in subsequent posts.

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. We will discuss them one by one.

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.

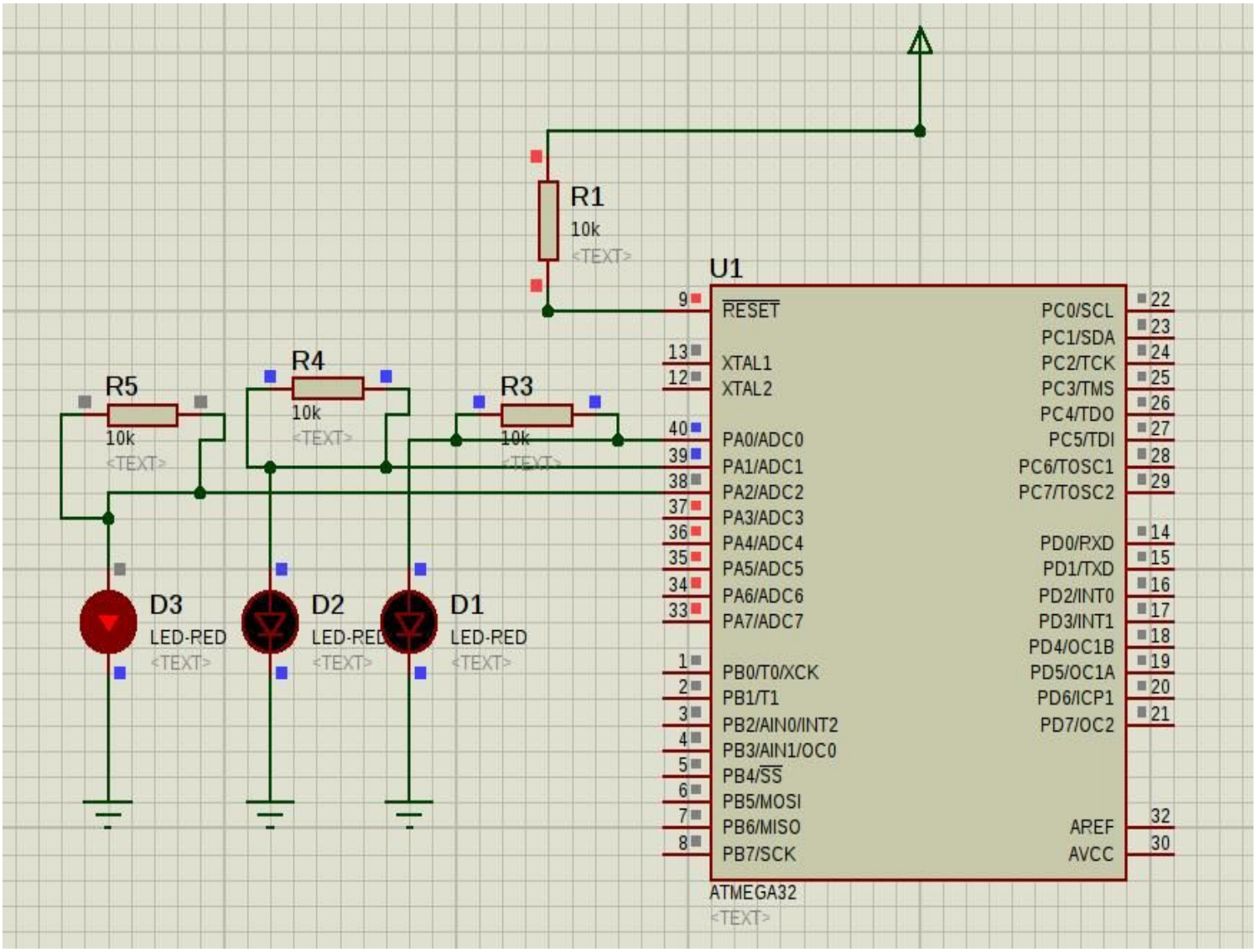
Choosing Prescalers

Let’s take an example. We need a delay of 184 ms (I have chosen any random number). We have F_CPU = 4 MHz. The AVR offers us the following prescaler values to choose from: 8, 64, 256 and 1024. A prescaler of 8 means the effective clock frequency will be F_CPU/8. Now substituting each of these values into the above formula, we get different values of timer value.

Prescaler for timer:

CS02	CS01	CS00	Description
0	0	0	Timer stoped
0	0	1	FCPU
0	1	0	FCPU/8
0	1	1	FCPU/64
1	0	0	FCPU/256
1	0	1	FCPU/1024
1	1	0	External Clock Source on PIN T0.Clock on falling edge
1	1	1	External Clock Source on PIN T0.Clock on rising edge

Result :



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

Appendix:

```
/*
* 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);

}
```

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

// ***************************************************************************
// Interrupt Routines
// ***************************************************************************
uint32_t counter = 0;
```

```
// timer0 overflow ISR(TIMER0_OVF_vect) {




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

// ***************************************************************************
// Main
// ***************************************************************************
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) {
}
```