Ministerul Educației al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare,Informatică și Microelectronică

Laboratory work nr.5

REPORT

At Embedded systems

“Interrupts , Timers and Tasks”

st. gr. FAF-141: Cristea Victor

Verified by: Andrei Bragarenco

Chişinău 2016

Goal of the work:

- Studying and understanding the timer
- Implement a basic task scheduler using timer
- Understanding the principle of ADC conversion
- Understanding LCD working principles

Condition

Write a C program and schematics for task runner which will run a task in defined interval of time with defined run properties.

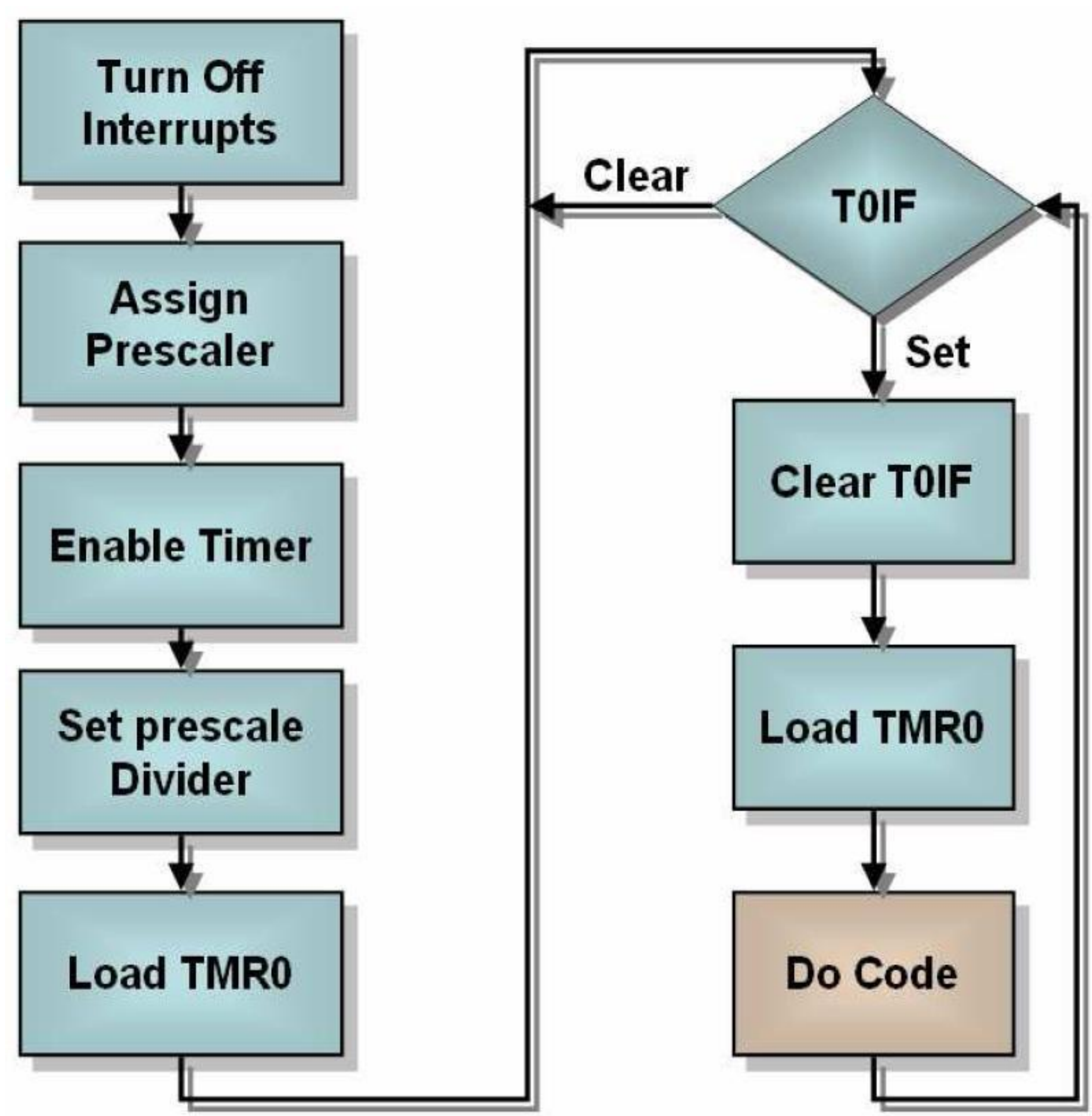
Theory

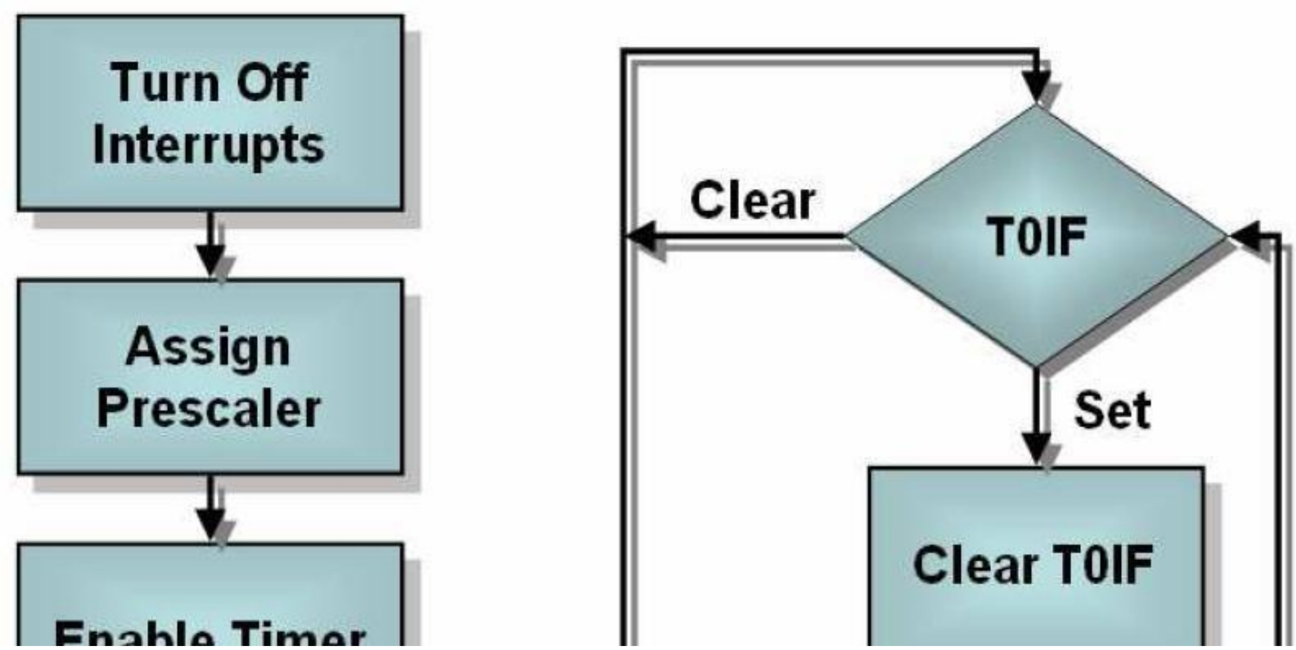
Basics of the timers

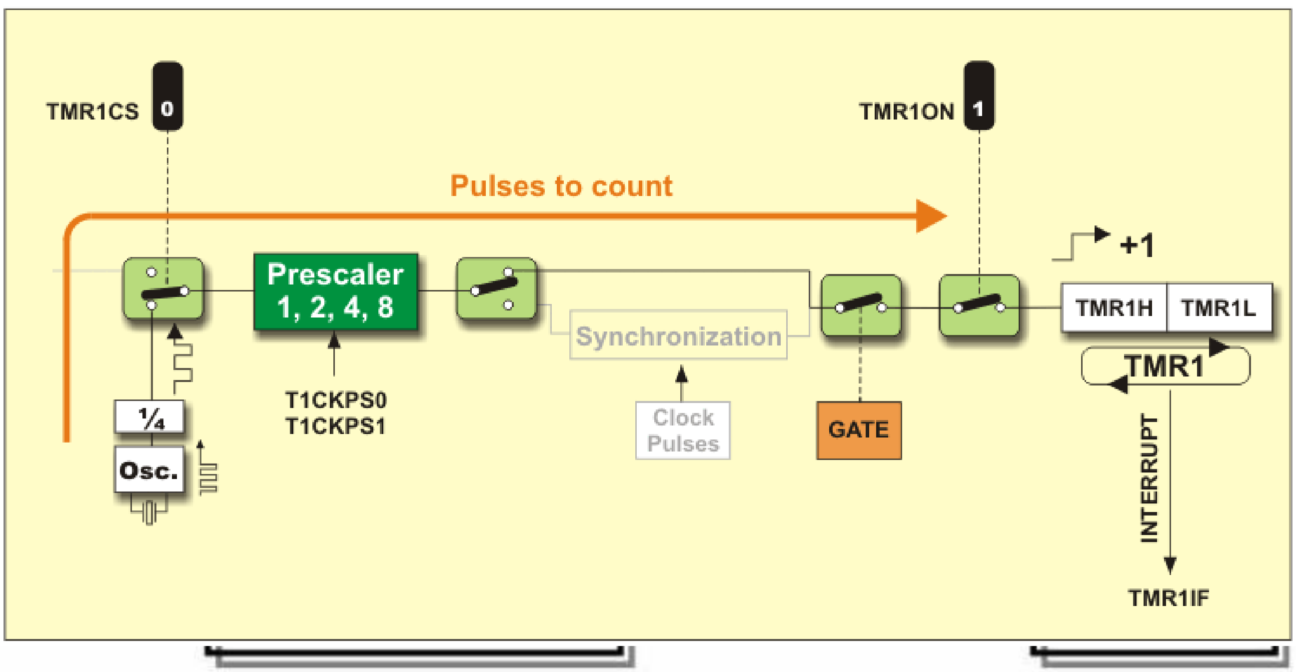
A typical timer will consist of a prescaler, an N-bit timer/counter register (typically N is 8, 16 or 32 bits), one or more N-bit capture registers, and one or more N-bit compare registers. There will also be control and status registers to configure and monitor the timer. The fundamental hardware involved is an up-counter, which counts incoming pulses. A counter, any counter, becomes a timer when the incoming pulses are at a fixed, known frequency. I will often use the terms “counter” and “timer” interchangeably, but just keep in mind that the hardware is always a counter, and that the counter becomes a timer if we are feeding it a fixed, known frequency to count.
Something else that you may see are uCs that have multiple timers of different size. For example, most AVRs have both 8-bit and 16-bit timers, and some LPC2xxx devices have both 32-bit and 16-bit timers.

Timers

Just about every microcontroller comes with one or more (sometimes many more) built-in timer/counters, and these are extremely useful to the embedded programmer - perhaps second in usefulness only to GPIO. The term timer/counter itself reflects the fact that the underlying counter hardware can usually be configured to count either regular clock pulses (making it a timer) or irregular event pulses (making it a counter). This tutorial will use the term “timer” rather than “timer/counter” for the actual hardware block, in the name of simplicity, but will try and make it clear when the device is acting as an event counter rather than a normal timer. Also note that sometimes timers are called “hardware timers” to distinguish them from software timers which are bits of software that perform some timing function.







As an example of a fixed selection prescaler, many AVR devices allow for fixed prescale values of 1, 8, 64, 256 and 1024. If the system clock frequency is e.g. 8MHz, this results in a timer clock period of 0.125, 1, 8, 32 or 128 microseconds per clock tick. Likewise, the 9S12 prescaler allows fixed prescale values in powers of 2: 1, 2, 4, 8, 16, 32, 64 and 128.

As an example of a fully configurable prescaler, the NXP LPC2000 family has 32-bit timers that have 32-bit prescalers, which allow for any prescale value between 1 and 2^32. Such fully configurable prescalers are of course more flexible than fixed selection prescalers, but in practice the fixed selection prescalers are usually adequate, though they may require that more thought be given to choosing a system clock frequency and other timing values.

Interrupts

An interrupt is a signal (an "interrupt request") generated by some event external to the CPU , which causes the CPU to stop what it is doing (stop executing the code it is currently running) and jump to a separate piece of code designed by the programmer to deal with the event which generated the interrupt request. This interrupt handling code is often called an ISR (interrupt service routine). When the ISR is finished, it returns to the code that was running prior to the interrupt, which then resumes running with no awareness that it has been pre-empted by the interrupt code. It is this ability to run the appropriate code for an external event at any point in time that is both the chief benefit of, and the potential source of difficulties from, interrupts.

A computer interrupt can be compared to interruptions in everyday life. For example, the ringing of your telephone or a knock on your door are interrupt-type events. Your phone or your doorbell can ring at any point during your day, and when they do ring you will typically stop what you are doing, deal with the reason behind the phone call or front door visit, and then go back to what you were doing. Likewise, just as you can ignore a phone call or doorbell if what you are doing requires your immediate and undivided attention, so it is possible to program the CPU to defer interrupts during certain critical sections of code, or even to ignore them entirely.

Interrupts are one of the most powerful and useful features one can employ in embedded systems. They can make the system more efficient and more responsive to critical events, and they can also make the software easier to write and understand. However, they can also be a confusing and error-prone feature in a program, and some people avoid them for that reason. But every embedded programmer should be at home with interrupts, should consider them as useful tools and not monsters in the closet, and use them whenever appropriate. It is fairly uncommon that an embedded system does not use at least one interrupt.

Solving

Timer.c

```c
#include
"timer.h"
            // initialize timer, interrupt and variable
            void timer0_init()
            {
                // set up timer with prescaler = 256
                TCCR0 |= (1 << CS02);

                // initialize counter
                TCNT0 = 0;

                // enable overflow interrupt
                TIMSK |= (1 << TOIE0);

                // enable global interrupts
                sei();

                // initialize overflow counter variable
                tot_overflow = 0;
            }
```

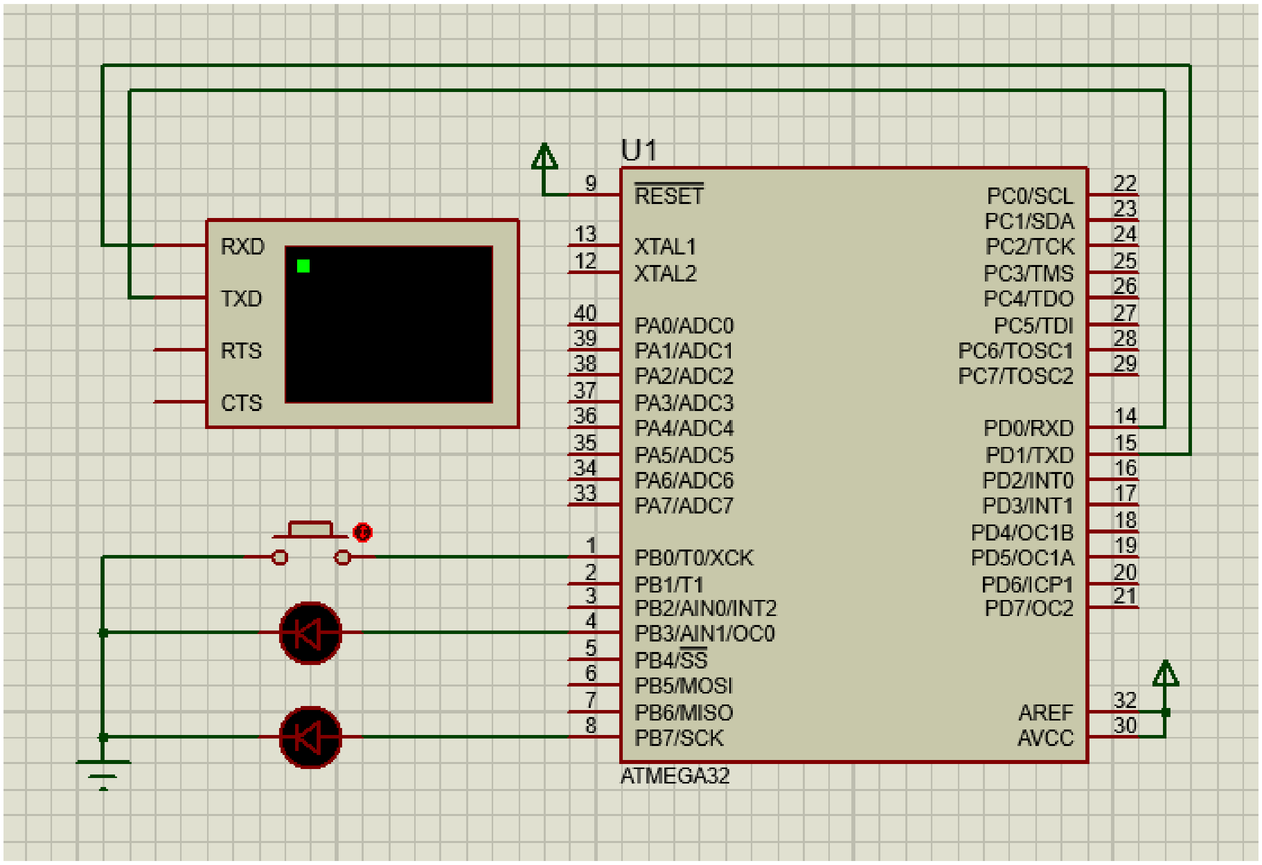
Timer.h

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

             // global variable to count the number of overflows
             volatile uint8_t tot_overflow;

             // initialize timer, interrupt and variable
             void timer0_init();
```

Schematics



Conclusion

In this laboratory work I learnt more concepts about microcontroller programming in C language and I designed a circuit in Proteus using timers and that timers are crucial component of most embedded systems

Appendix

main.c

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


// Interrupt Routines

uint32_t counter = 0;

// timer0 overflow
ISR(TIMER1_OVF_vect) {
	// XOR PORTA with 0x01 to toggle the second bit up
	turn_off(0);
	turn_off(1); turn_off(2);
	turn_off(3); turn_off(4);
	turn_off(5);
	toggle_led(counter); counter++;

	if(counter >= 6) { counter = 0;
	}


}

int main( void ) {
	1.Configure  PORTA  as  output  DDRA  =
	0xFF;
	PORTA = 0x00;

	2.enable timer overflow interrupt for both Timer0 and Timer1
	TIMSK=(1<<TOIE0) | (1<<TOIE1);
	3.set timer0 counter initial value to 0
	TCNT1=0x00;

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

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);}
```

led.h

```
#ifndef LED_H_ #define 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
```