Experiment-4

Aim: Hands-on experimentation of ATMega32Timer with interrupt programming in C.

Objectives: After successfully completion of this experiment students will be able to,

- Use C language for ATMega32 microcontroller programming on AVRStudio.
- Experiment with timer with interrupt of ATMega32 on ATMega32 AVR Development Board.

Equipment required:

- Windows7 or later based host computer
- ATMega32 Development board
- USBasp Programmer
- Jumper Wires
- LED

Software required:

- AVR Studio7 installation setup
- USBasp driver installation setup

Theory:

AVR Interrupt Timer programming Basics

In this section, we'll take a look at interrupts, how we deal with them under AVR, and their impact on scheduling and real-time.

The first thing we need to ask is, "What's an interrupt?"

An interrupt is exactly what it sounds like — an interruption of whatever was going on and a diversion to another task.

For example, suppose you're sitting at your desk working on job "A." Suddenly, the phone rings. A Very Important Customer (VIC) needs you to immediately answer some skill-testing question. When you've answered the question, you may go back to working on job "A," or the VIC may have changed your priorities so that you push job "A" off to the side and immediately start on job "B."

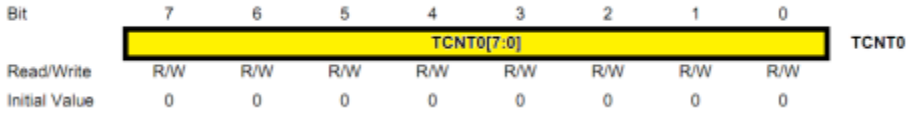
In most microcontrollers, there is something called interrupt. This interrupt can be fired whenever certain conditions are met. Now whenever an interrupt is fired, the AVR stops and saves its execution of the main routine, attends to the interrupt call (by executing a special routine, called the Interrupt Service Routine, ISR) and once it is done with it, returns to the main routine and continues executing it.

For example, in the condition of counter overflow, we can set up a bit to fire an interrupt whenever an overflow occurs. Now, during execution of the program, whenever an overflow occurs, an interrupt is fired and the CPU attends to the corresponding ISR. Now it's up to us what we want to do inside the ISR. We can toggle the value of a pin, or increment a counter, etc etc.

If you didn't get the concept of interrupts and ISR, behold for sometime till we discuss it how to implement it in hardware.

TCNT0 Register

The Timer/Counter Register – TCNT0 is as follows:



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.

Now we know where the counter value lies. But this register won't be activated unless we activate the timer! Thus we need to set the timer up. How? Read on…

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.

Clock Select Bit Description

we choose No Prescaling. Ignore the bits highlighted in grey. Thus, we initialize the counter as:

```
TCCR0 |= (1 << CS00);
```

Please note that if you do not initialize this register, all the bits will remain as zero and the timer/counter will remain stopped.

Methodology – Using Interrupts

So now, we have to flash the LED every 1000 ms. With CPU frequency 1 MHz, even a maximum delay of 1.024 ms can be achieved using a 1024 prescaler. So what do we do now? Well, we use interrupts.

The concept here is that the hardware generates an interrupt every time the timer overflows. Since the required delay is greater than the maximum possible delay, obviously the timer will overflow. And whenever the timer overflows, an interrupt is fired. Now the question is *how many times should the interrupt be fired?*

For this, let's do some calculation. Let's choose a prescaler, say 256. Thus, as per the calculations, it should take 65.280 ms for the timer to overflow. Now as soon as the timer overflows, an interrupt is fired and an Interrupt Service Routine (ISR) is executed. Now,

1000 ms ÷ 65.280 ms = 15.3186ms

Thus, in simple terms, by the time the timer has overflown 15 times, 979.2 ms would have passed. After that, when the timer undergoes 16th iteration, it would achieve a delay of 979.2 ms. Thus, in the 16th iteration, we need a delay of 1000 – 979.2 = 20.8 ms. At a frequency of 3.90 kHz (prescaler = 256), each tick takes 0.256 ms. Thus to achieve a delay of 20.8 ms, it would require 81 ticks. Thus, in the 16th iteration, we only allow the timer to count up to 81, and then reset it. All this can be achieved in the ISR as follows:

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

// TIMER0 overflow interrupt service routine
// called whenever TCNT0 overflows
ISR(TIMER0_OVF_vect)
{
   // keep a track of number of overflows
   tot_overflow++;
}

int main(void)
{
   // connect led to pin PC0
   DDRC |= (1 << 0);
```

```
   // initialize timer
   timer0_init();

   // loop forever
   while(1)
   {
      // check if no. of overflows = 15
      if (tot_overflow >= 15)  // NOTE: '>=' is used
      {
         // check if the timer count reaches 81
         if (TCNT0 >= 81)
         {
            PORTC ^= (1 << 0);        // toggles the led
            TCNT0 = 0;           // reset counter
            tot_overflow = 0;   // reset overflow counter
         }
      }
   }
}
```

Please note that the code is not yet ready. Not until you learn how to enable the interrupt feature. For this, you should be aware of the following registers.

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	

TIMSK Register

TIFR Register

The Timer/Counter Interrupt Flag Register– TIFR is as follows. Even though we are not using it in our code, you should be aware of it.

Bit	7	6	5	4	3	2	1	0	
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
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	

TIFR Register

This is also a register shared by all the timers. Even here, bits 1 and 0 are allotted for TIMER0. At present we are interested in the 0th bit TOV0 bit. This bit is set (one) whenever TIMER0 overflows. This bit is reset (zero) whenever the Interrupt Service Routine (ISR) is executed. If there is no ISR to execute, we can clear it manually by writing one to it.

In this example, since we are using ISR, we need not care about this bit (thus this register as a whole).

Enabling Global Interrupts

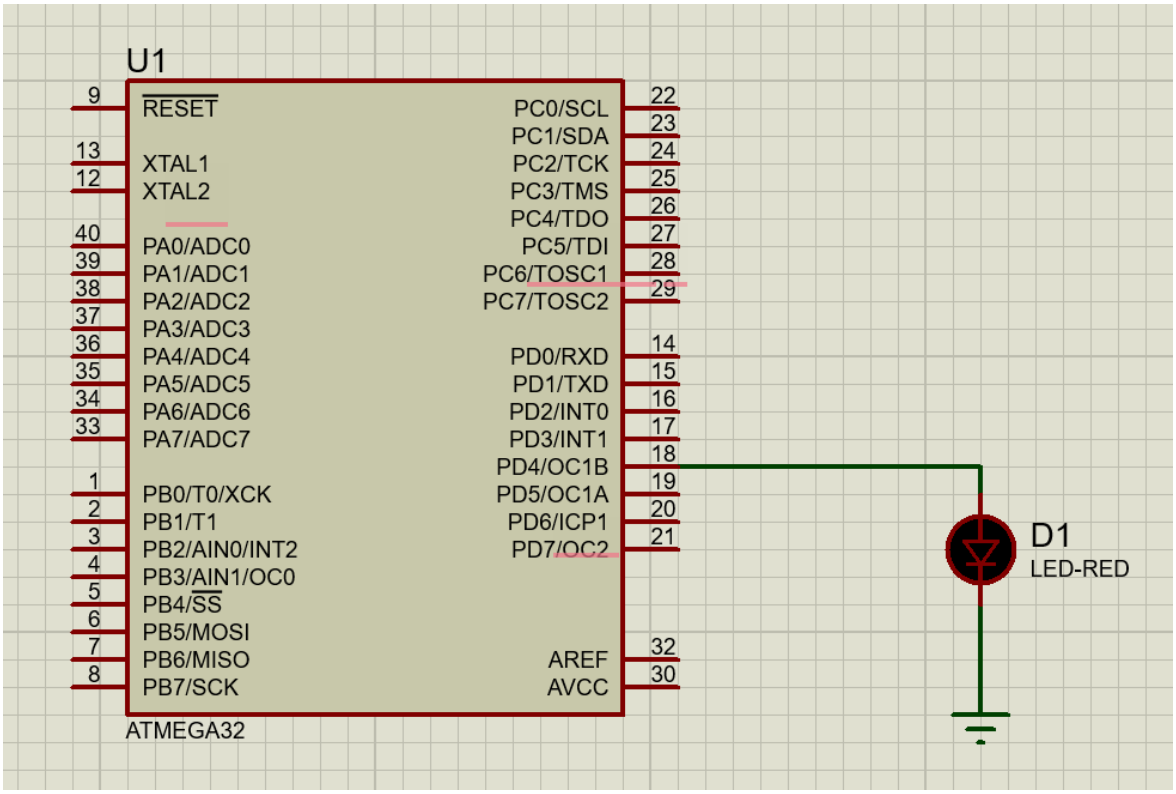
In the AVRs, there's only one single bit that handles all the interrupts. Thus, to enable it, we need to enable the global interrupts. This is done by calling a function named sei(). Don't worry much about it, we simply need to call it once, that's all.

```
#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()
{
   // 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;
}

// TIMER0 overflow interrupt service routine
// called whenever TCNT0 overflows
ISR(TIMER0_OVF_vect)
{
   //Keep a track of number of overflows
   tot_overflow++;
}

int main(void)
```

```
{
   // connect led to pin PC0
   DDRC |= (1 << 0);

   // initialize timer
    timer0_init();

   // loop forever
   while(1)
   {
      // check if no. of overflows = 15
      if (tot_overflow >= 15)  // NOTE: '>=' is used
      {
         // check if the timer count reaches 81
         if (TCNT0 >= 81)
         {
            PORTC ^= (1 << 0);        // toggles the led
            TCNT0 = 0;           // reset counter
            tot_overflow = 0;   // reset overflow counter
         }
      }
   }
}
```

OUTPUT:



CONCLUSION:

By performing this experiment I came to know about the interrupt programming of AVR

Experiment-4 **Post Lab Exercise**

Student Name: Aryan Langhanoja
Enrollment No: 92200133030

Answer the following questions:

1) What is the role of the TIMSK register?

- TIMSK Stands for Timer/Counter Interrupt Mask Register which is used to enable or enable or disable specific timer/counter interrupts.

2) What is the role of the TIFR register?

- The TIFR register (Timer/Counter Interrupt Flag Register) is used to indicate the occurrence of timer/counter-related events and interrupts.

3) OCF1A comes in which mode Normal mode or CTC mode? In the above code of interrupt, which mode we have used normal or CTC?

- OCF1A comes in CTC Mode. In the above code, we used a Normal Mode.

4) Assume the XTAL = 8 MHz. Find the TCNT0 value needed to generate a time delay of 5000 ms. Use normal mode and the largest prescaler possible.

- We cannot generate a 500 ms delay with Time0. Because imer0 can generate a maximum delay of 32ms.

5) Assume the XTAL = 1 MHz. Find the OCR0 value needed to generate a time delay of 20 ms. Use CTC mode, no prescaler.

- We cannot generate 20 ms of delay with given credentials.