Lab 6: AVR Interrupts

EE222: Microprocessor Systems

Contents

1 Administrivia

1.1 Learning Outcomes

By the end of this lab you will be able to:

1. Code timer interrupts for efficient scheduling in ATmega16A.

1.2 Deliverables

You are required to submit

- Appropriately commented Source Code
- Experimental observations and problems you faced

in the beginning of next lab.

2 Hardware Resources

- ATmega16A Microcontroller Unit
- Universal Programmer
- Seven Segment Display
- Resistances 47Ω
- Switch (You may use from the trainer kit)
- Led (You may use from the trainer kit)

3 Interrupts

In previous labs we implemented delays by polling the timer overflow flag. Although accurate, but this still is a process that consumes all the processing time (and energy) just waiting for a flag. In this lab we are going to use interrupts. Interrupts are implemented through special block of hardware integrated inside the processing device. Whenever some specified event occurs, the hardware halts the normal execution of code and move instruction pointer to a reserved block of code called interrupt service routine and the process is called interrupting the processor. To understand how this happens and in what basic forms the technique can be exploited, we will use the timer interrupts in this lab.

3.1 Timer Interrupts

There are eight interrupts associated with timers. Whenever an event from the list below occurs,

1. The hardware halts normal execution of code.
2. Pushes the address of next instruction on stack.
3. Jumps to Program memory location specified in the table below.
4. Execute the code written there.
5. After successful execution pops back the address of instruction and continue normal execution.

Interrupt	**Program Memory Location**
Timer 2 Compare Match	`0x0008`
Timer 2 Overflow	`0x000A`
Timer 1 Capture Event	`0x000C`
Timer 1 Compare Match A	`0x000E`
Timer 1 Compare Match B	`0x0010`
Timer 1 Overflow	`0x0012`
Timer 0 Compare Match	`0x0014`
Timer 0 Overflow	`0x0016`

3.2 Understanding the merits of interrupts

Suppose we have to design a system such that two independant tasks are to be performed at the same time. One is to toggle an led with 4Hz frequency and other is to relay the state of a switch to another led.
If we use a delay function to produce 250ms delay, our microcontroller will get busy polling over the overflow flag (or Compare match flag if using CTC mode) and it will not be able attend the transitions in state of switch for about 250ms.
Interrupts can rescue us from such situations. Let us understand the code below,

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

#define LED01   0     // defining led 1 location
#define LED02   1     // defining led 2 location
#define SW      7     // definig switch location

int main()
{
    /*
     * Setting the pin number specified by macros
     * LED01 and LED02 of port B as output pins.
     *
     * Pin specified by macro SW will naturally
     * be set as input.
     */
    DDRB = (1<<LED01)|(1<<LED02);
    PORTB = (1<<LED01); // turning on led 1

    TCNT0 = 12;      // initializing timer
    TCCR0 = 0x05;    // setting prescalar clk / 1024
    sei();           // enable global interrupt
    TIMSK = 0x01;    // enable timer 0 overflow interrupt
    while(1)
    {
        if( (PINB & (1 << SW) ) != 0) // check if switch is high
        {
            PORTB |= (1<<LED02); // turn on led 2
        }
        else
        {
            PORTB &= ~(1<<LED02); // turn off led 2
        }
    }
}

ISR(TIMER0_OVF_vect)
{
    TCNT0 = 12;                  // reload timer
    PORTB = PORTB ^ (1<<LED01); // toggle led 1
}
```

We are using clock of 1MHz with 1024 prescalar,

$$Time\ for\ one\_tick = 1024/1M = 1024\mu s = 1.024ms$$

$$Ticks\ for\ 250ms = 250/1.024 = 244$$

$$Value\ to\ be\ loaded\ in\ TCNT0 = 256 - 244 = 12$$

“sei()” function defined in **“avr/interrupt.h”** sets the gloabal interrupt enable bit. If this bit is not set, no interrupt will be responded to.
`TIMSK` (Timer Interrupt Mask) register contain bits to enable individual timer interrupts.

OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0

`TIMSK` **bit**	**Function**
OCIE2	When 1, enables Timer 2 Compare Match interrupt
TOIE2	When 1, enables Timer 2 Overflow interrupt
TICIE1	When 1, enables Timer 1 Input Capture interrupt
OCIE1A	When 1, enables Timer 1 Compare Match A interrupt
OCIE1B	When 1, enables Timer 1 Compare Match B interrupt
TOIE1	When 1, enables Timer 1 Overflow interrupt
OCIE0	When 1, enables Timer 0 Compare Match interrupt
TOIE0	When 1, enables Timer 0 overflow interrupt

Note that the number of timer interrupts is same as the number of timers related flags in `TIFR`.
In while loop the switch is being polled and its state is being relayed to led 2. In parallel with this process, timer is continuously being incremented and as it overflows, the timer 0 overflow flag is set. The processor is interrupted, it stops whatever it is executing at that moment (in the main function) and jumps to **"ISR(TIMER0_OVF_vect)"**, executes the code written there and then resumes from where it stopped the normal processing. All these steps are performed by hardware itself to make the life of programmer easy.
While the processor is attending an ISR (Interrupt Service Routine), the flag which was the cause of that interrupt (like `TOV0` in example above) is automatically cleared by the processor.
Service Routines for other timer interrupts can be written by passing following macros (vector names) defined in "avr/interrupt.h" as an argument to ISR.

Interrupt	**Vector name**
Timer 2 Compare Match	TIMER2_COMP_vect
Timer 2 Overflow	TIMER2_OVF_vect
Timer 1 Capture Event	TIMER1_CAPT_vect
Timer 1 Compare Match A	TIMER1_COMPA_vect
Timer 1 Compare Match B	TIMER1_COMPB_vect
Timer 1 Overflow	TIMER1_OVF_vect
Timer 0 Compare Match	TIMER0_COMP_vect
Timer 0 Overflow	TIMER0_OVF_vect

```
ISR ( . . . . . )
{
        . . .
        . . .
}
```

More than one ISRs can be written in a program but for different interrupts (not for the same interrupt). When the processor is servicing one interrupt, it ignores other interrupts.

4 Lab Tasks

4.1 Task A

1. Connect a seven segment display “SSEG” with your ATmega16A.
2. Connect an led “LED” and a switch “SW” with your ATmega16A.
3. The SSEG should display continuous counting from `0-9` and then again from `0`, each digit with a 1 second delay.
4. At the same time, the state of SW should be relayed to LED (–that is, if SW is high, LED should be on and vice versa).
5. Note that both of the tasks are completely independent and should be performed in parallel without lag.