Tutorial 4

Interrupts for AVR Microcontroller

# Timer Interrupt

In this task, we will get used to the concept of Timer Interrupt. To be specific, we will use the **Overflow (Timer) Interrupt** generated by the Timer 1 (16-bit). Our main task is to toggle a pin when the interrupt happens.

## Calculation

We are going to use the 16-bit Timer 1 in this task. The timer is **overflow** when it reaches the maximum value of **65535** (216-1), in turn triggers an overflow interrupt. Every interrupt request receives, we will toggle the pin with a frequency determined as follows:

With the Target Timer Count = TOP Value = 65535, we will calculate back the frequency:

In this task, we choose the **maximum prescaler of 64**, so we can estimate the blinking frequency:

## Programming and Testing

Note – in this part we will develop the code step by step (so please read carefully)

1. Make a new project and type the following code.



If you compile and download the code onto the Board, you shall see nothing just yet

1. Besides the main, when the interrupt occurs, we need an interrupt routine to process the interrupt. Add in the following routine (highlighted in yellow)



1. Build and download code onto the board this time, you shall see the LED blinking **this time**. Observe the LED blinking with observable frequency **around 2 Hz.**

## Analysis

The explanation for the above code is as follows:

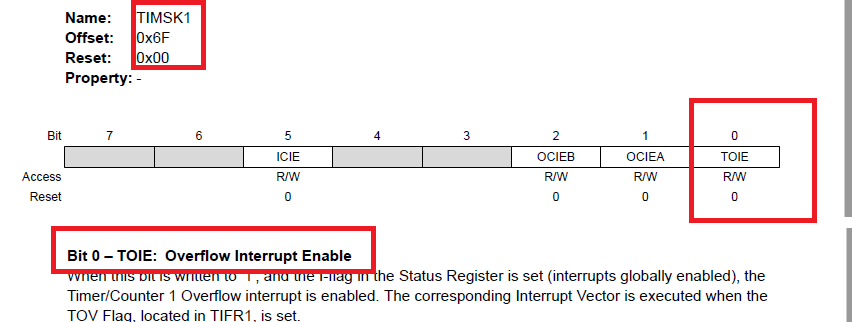
1. For this tutorial, we will use Interrupts hence the very first step is to include the corresponding header file

|  |
| --- |
| #include <avr/interrupt.h> |

1. First, you need to set for other operation, since we are using the Timer hence it is important to configure Timer first: like clock source, prescaler etc. *This is exactly how you have done in Tutorial 3*

|  |
| --- |
| TCCR1A = 0x00;  TCCR1B = (1<<CS10)| (1<<CS11); // Set up timer prescaler of 64 |

1. Next, we need to enable the interrupt – this is something you must do for all interrupts
   1. Step 1 - Enable the bit for that specific interrupt. Checking the datasheet at Chapter Timer 1, you should see the register corresponding with this TIMSK1 and you need to enable TOIE bit.



*Note there is a bit of mismatch in the name between datasheet and header file so just use the following*

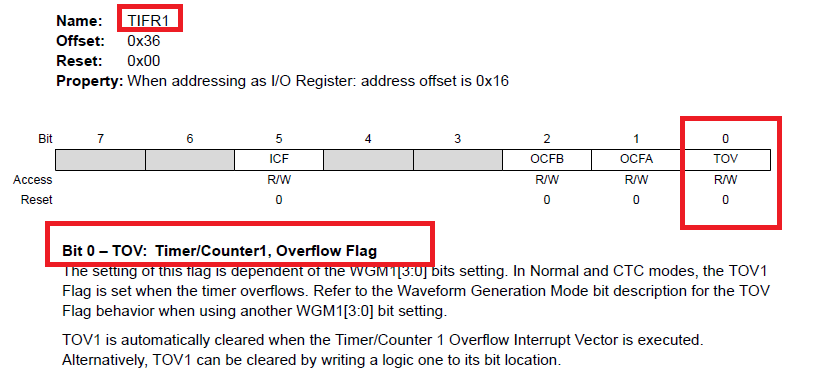
|  |
| --- |
| TIMSK1 = 1<<TOIE1; /\* Enable Overflow Interrupt \*/ |

* 1. Step 2 – Enable Global Interrupt Enable Bit of ISR

|  |
| --- |
| sei(); // // Enable global interrupts by setting global interrupt enable bit in SREG |

The two steps are essential for any interrupts.

1. For the timer overflow interrupt, to ensure the program is not affected by previous timer overflows (**which there is a possibility**) we will clear the overflow flag during the setup.



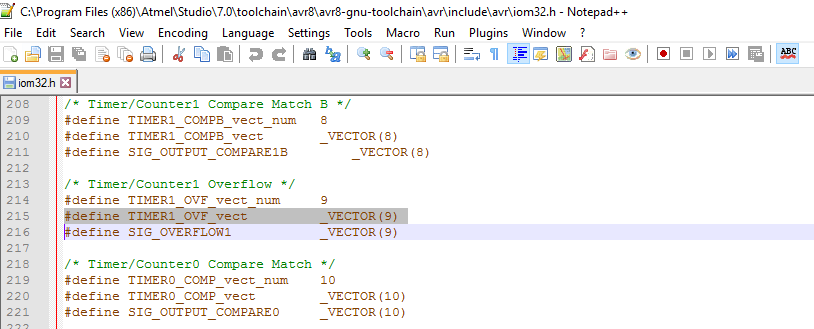
*Note there is a bit of mismatch in the name between datasheet and header file so just use the following*

|  |
| --- |
| TIFR1 = 1<<TOV1; /\* Clear overflow flag \*/ // For Atmega328p |

1. Now, we are ready to write the Interrupt Service Routine (ISR). If you notice, the main is kept empty at this moment hence we don’t want to do any other tasks for main, and just wait for the overflow happens or interrupt request to be sent.

|  |
| --- |
| while(1)  {  // Wow so code in the while loop. How come?  }  } |

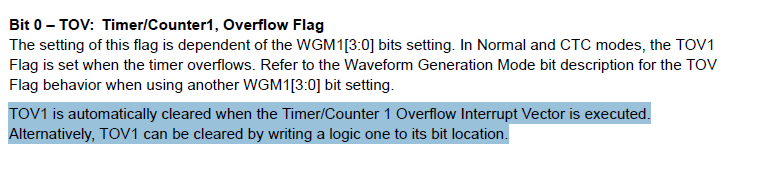
1. To develop the ISR, couple things we need to pay attention to:
   1. First you need to know the vector name, these vectors are declared in the io.h as you have included. *If you are asked to deal with different interrupts, you should check the vector name here*



* 1. Next, we will develop the main task, here we want to toggle a LED to see if the frequency is generated properly hence the PortB0 is toggled.

|  |
| --- |
| //Interrupt service routine  ISR (TIMER1\_OVF\_vect)  {  PORTB ^= (1 << PORTB5); /\* Toggle a pin on timer overflow \*/  } |

Note here – if you notice above, once the overflow occurs, we don’t have to clear the overflow flag TOV1 anymore (we set it once at the beginning just to clear previous interrupts, that is a different purpose). The reason is that the Flag will be automatically cleared if the Interrupt is served (done by Hardware)



# Clear Timer on Compare (CTC) Interrupt

In this task, we will work with the Clear Timer on Compare (CTC) mode of the AVR Timers and use CTC Interrupt. Specifically, once the Timers are in the CTC mode, if the counter value reaches a **preset** value (called *Compare value*), an Interrupt request will be sent, and then the counter value will be reset.

Note – as you might have done this in the Lab 1, you can use CTC mode without interrupt by manually check the flags and then clear it by software. In this part we will use Interrupt, which is Hardware-based approach

We will use this kind of interrupt to generate a 1Hz clock as we did in the **Tutorial 3 – Exercise 2**

## Calculation

A bit revision also on how we calculate the Timer Count value using the following formula:

From Exercise 2 in Tutorial 3

|  |  |
| --- | --- |
| **Prescaler Value** | **Target Timer Count** |
| 256 | 31,249 |

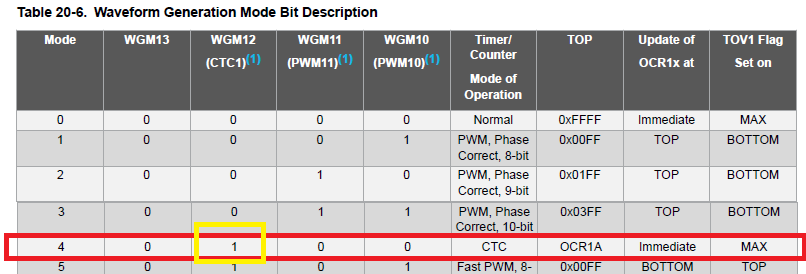
We used the prescale factor of 256 for Timer 1 and the timer counter value was 31,249, These two numbers will be used in CTC mode too.

## Programming and Analysis

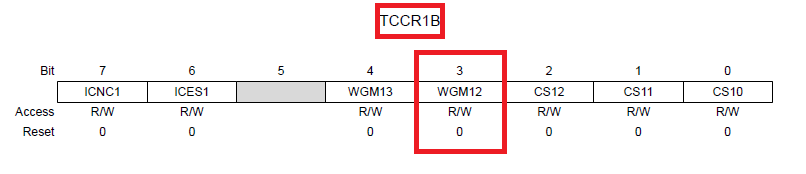
1. We will develop the program from the code in the Exercise 2 of Tutorial 3. Below is the code from that task**, which we will gradually change to work with Interrupts.**



1. First configure the Timer 1 to work in the CTC mode. To do so, we need to configure corresponding bits **WGMxx** in the **Timer Control Register A and B - TCCR1A and TCCR1B**.



So, we need to set the bit **WGM12** in the register **TCCR1B**



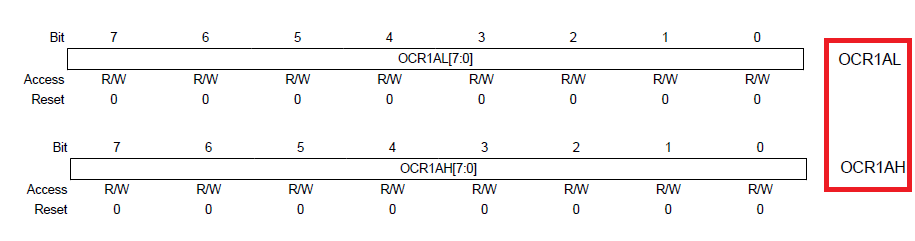
Put that in the code you can have the following

|  |
| --- |
| TCCR1B |= (1 << WGM12); // Turn on the CTC mode for Timer 1 |

Since the requirement is to generate 1Hz so we need to do prescaler of 256

|  |
| --- |
| TCCR1B |= (1 << CS12 ); // Set up Timer 1 with the prescaler of 256 |

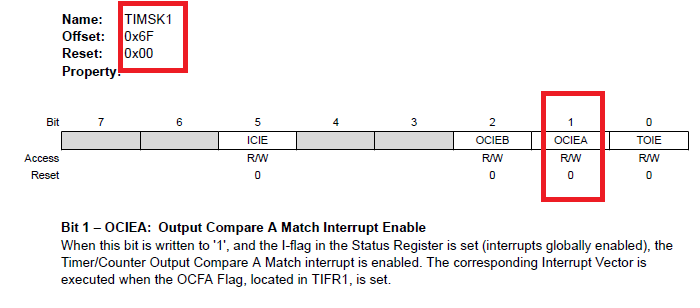
1. Next you need to set the **CTC compare value** by setting **Compare Register 1 A Register** (including High byte and Low Byte – because this is 16 bit long) – *Note for this we can set the decimal value in C*



|  |
| --- |
| OCR1A = 31249; // Set CTC compare value to 1Hz at 16 MHz AVR clock , with a prescaler of 256 |

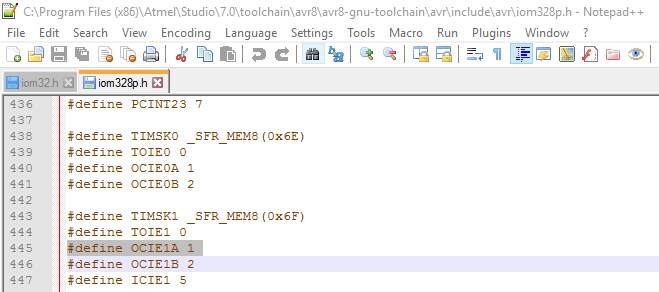
The value of desired timer count now is assigned to OCR1A

1. Enable interrupts:
   1. Step 1 - Next, we need to enable the CTC interrupt. In this task, we will use only one Compare value so we will turn on the **Output Compare A Match Interrupt Enable - OCIEA** in the **Timer Interrupt Mask Register - TIMSK1.**



Note here – Even though in the datasheet said the pin named is OCIEA, if you use it in the code, it will give you the error message.

If you check the header file, it still lists as OCIE1A



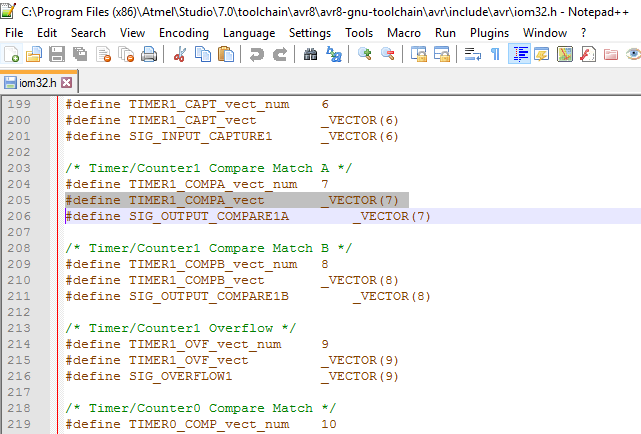
Hence, we are still using the as follows (This is one of the rare cases where we got a mismatch between the Datasheet and the IDE so we need to trace back to the source)

|  |
| --- |
| TIMSK1 = 1<<OCIE1A; // Enable Output Compare A Match Interrupt |

* 1. Step 2 – to turn on the global interrupt

|  |
| --- |
| sei(); //Enable the Global Interrupt Bit |

1. Next, we will need to write the ISR as done in the Task 1
   1. Check the vector name for the Interrupt



* 1. Write the ISR, since we need to check the frequency hence a simple toggling of a IO port is enough

|  |
| --- |
| ISR ( TIMER1\_COMPA\_vect )  {  PORTB ^= (1 << 5); // Toggle the LED  } |

1. The final code looks like below:



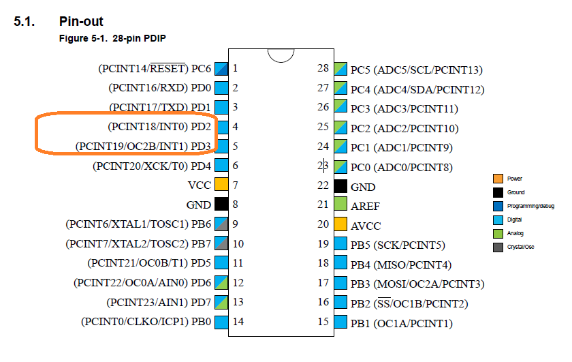
1. Build and download the code to the board.
2. Observe the blinking frequency to see it is around 1Hz.

# Basic External Interrupt

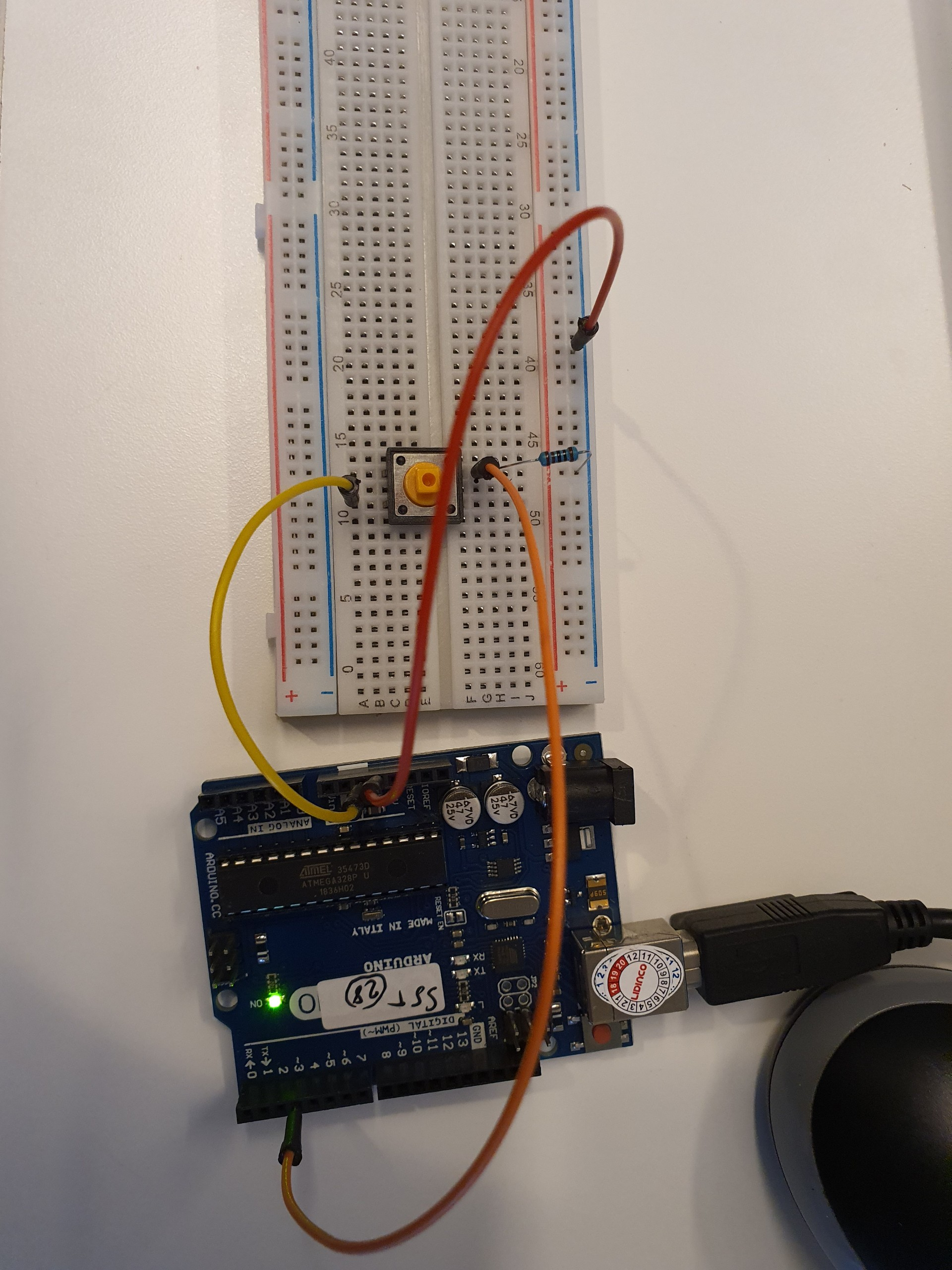
In this task, we will get started with **External Interrupt**, which detect changes externally of a GPIO pin. As the request receive, we will then light up one LED to indicate this.

## Hardware Setup

1. Connect a button (don’t forget Pull up or Pull down) to the External Interrupt Source pin 0 INT0 (Pin 4), Pin PD2



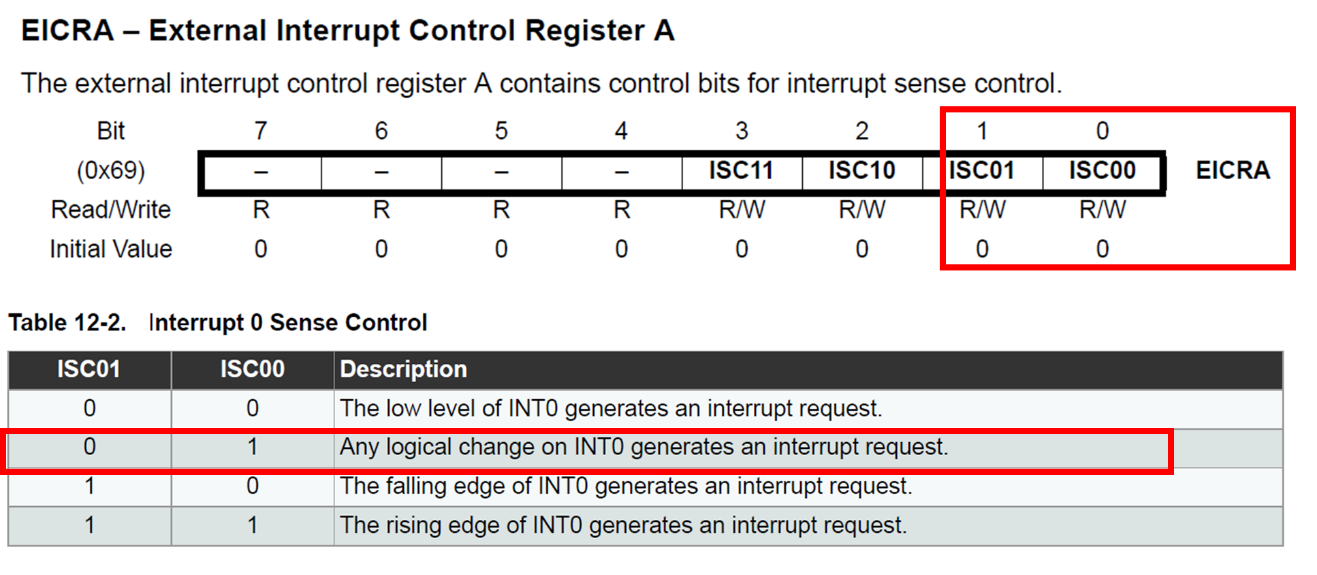
**The position of the pin is very important since this External Interrupt works with this specific pin**



**Check out the videos for more information.**

## Programming and Analysis

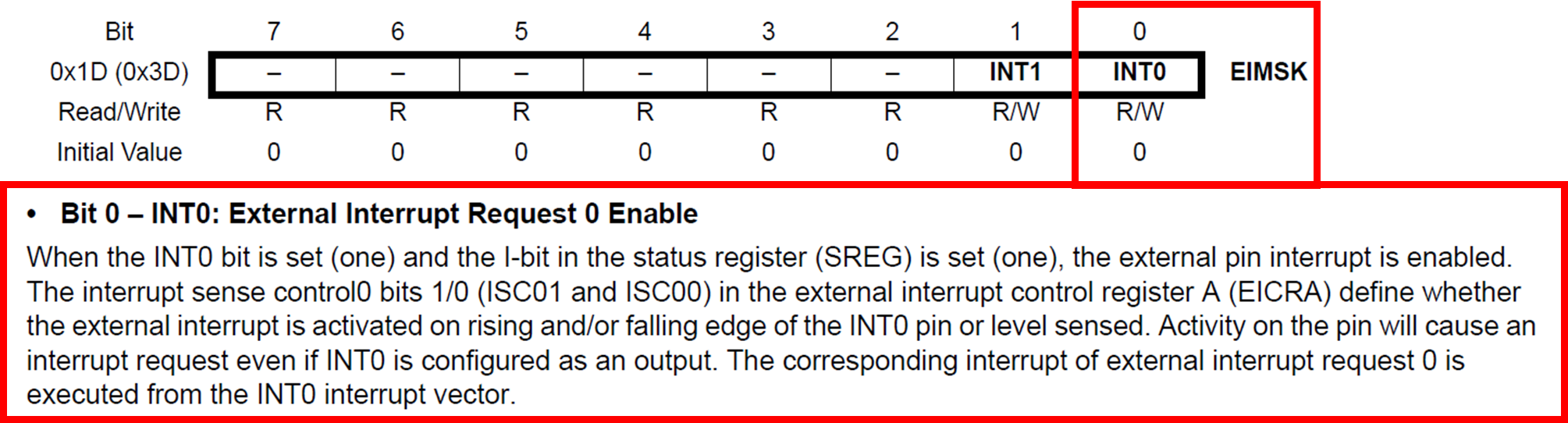
1. You need to first configure the Control Register, configure the **External Interrupt Control Register A – EICRA.** Check the datasheet from **Chapter 12 – External Interrupts**. At this moment let’s simply set if the pin detects any changes, an external interrupt request will be generated.



To do so we need to set the bit **ISC00** to **1**

|  |
| --- |
| EICRA |= (1 << ISC00); // set INT0 to trigger on ANY logic change |

1. Enable interrupts
   1. Step 1 - Enable the External Interrupt **External Interrupt Mask Register - EIMSK**

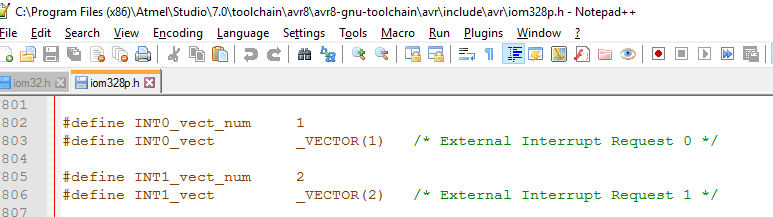


To do so we need to turn on the bit INT0

|  |
| --- |
| EIMSK |= (1 << INT0); // Turns on INT0 |

* 1. Step 2 - Next you can enable the **Global Interrupt Enable Bit** by **sei()**

1. Next for ISR
   1. Check the name of the vector



* 1. Write a simple ISR to toggle the LED

|  |
| --- |
| ISR(INT0\_vect) // ISR for the INT0  {  PORTB ^= (1<<PORTB5); //Toggle the PB5 connecting to the LED  } |

1. The full code is as follows:

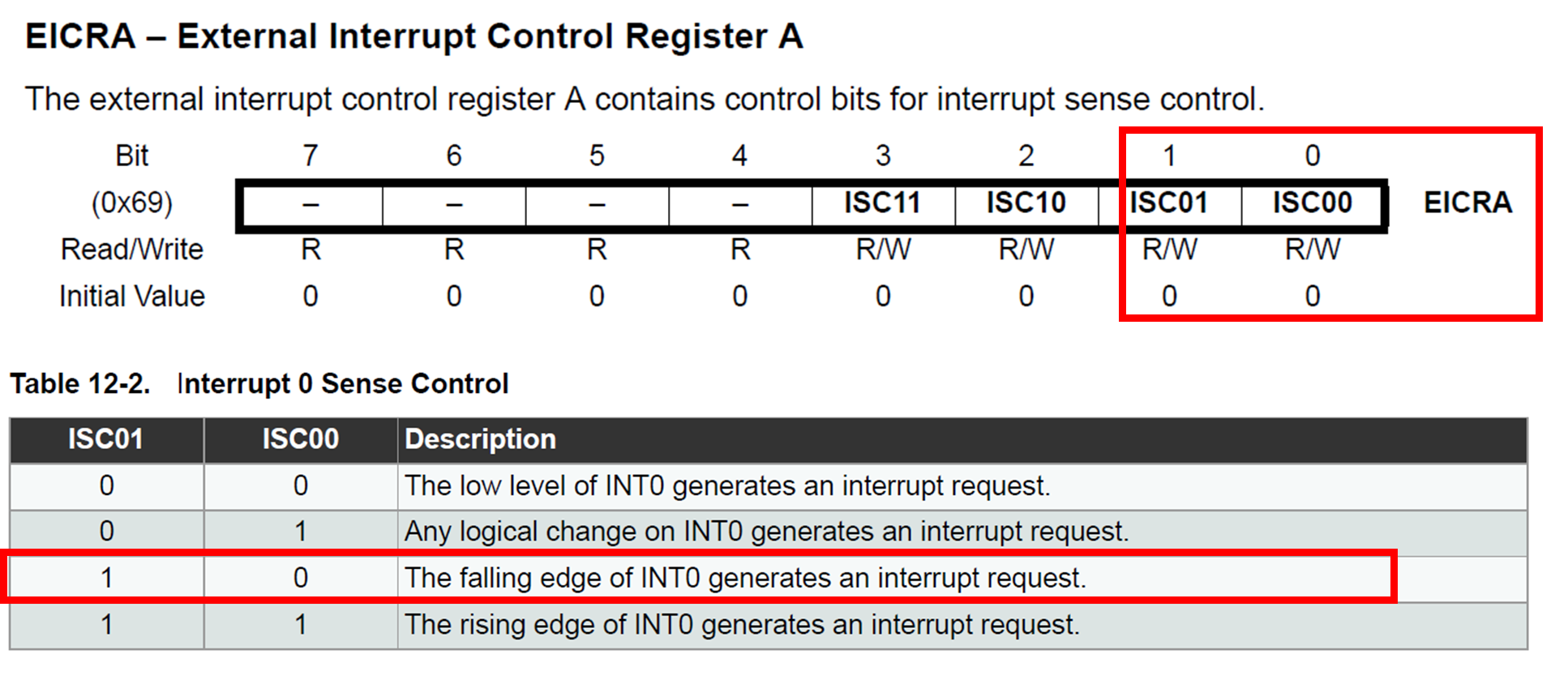


1. Build and download the code to the board.
2. Press and hold the button and observe if the LED is turned ON.
3. Release the button to see if the LED turned OFF.

The above behaviour due to the setting for the Interrupt Sense Control that *“Any logical change on INT0 generates an interrupt request.”*

*Note – if at this stage, you don’t see the button working stable – this might be due to bad connection, bad switch and bouncing buttons so don’t panic. Try to replace the button first and try again first*

1. Change the sense control to sense the falling edge by configuring again the **EICRA** register



1. The modified code is



1. Rebuild and download the code to the board.
2. Press and release the button, observe if the LED is **toggled once.**

If you observe this, it means the MCU has receive the request properly and response to it