Topic 9:
Debouncing, Timers and Interrupts

Luis Mejias

Outline

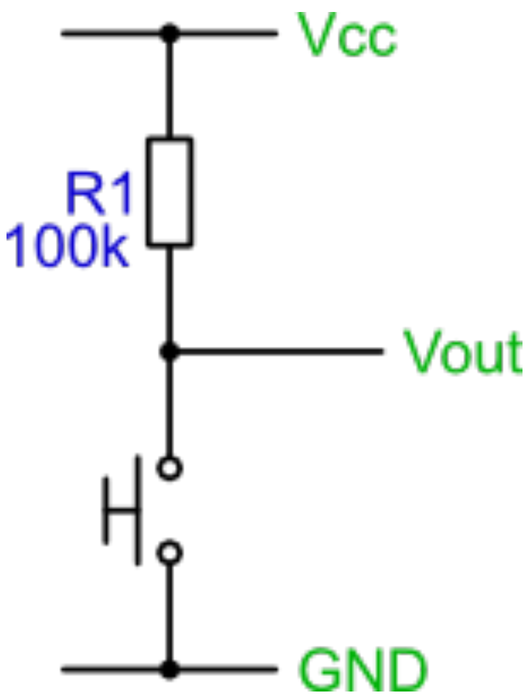
- Review Topic 8: UART
- Debouncing
- Timers
- Interrupts
- Examples

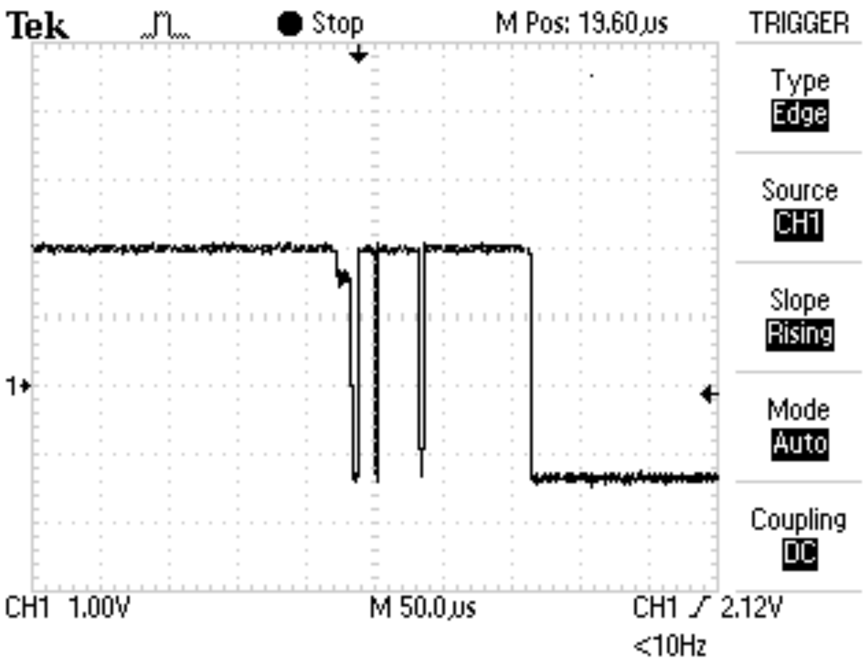
Review Topic 8: UART

Debouncing

- A switch is a mechanical component, and as a result there is often mechanical noise caused by contact bouncing.
- Contact bouncing is a physical problem that arises when the switch's contacts come together. This connection is not instantaneous and consequently noise is generated.
- This noise causes bouncing, which means that the output from the switch bounces between logical high (i.e. the high voltage) and logical low (i.e. the low voltage which is often ground).
- The results is that a single press appear like multiple presses causing the pin we are interested in reading going rapidly and repeatedly between a circuit's high voltage state and its low voltage state.
- Fortunately this can be overcome through switch debouncing, which is typically performed in software implementations (can also be performed at the hardware level).
- In both software and hardware there are multiple ways of dealing with the problem, as is explained on the second page of this link: http://www.ganssle.com/debouncing.htm. The hardware provided for this unit does not have software bouncing in the physical circuitry and consequently debouncing will have to be implemented with software. Switch debouncing is an extremely important component of reading switch measurements in embedded systems.

Pull-up resistor example





Debouncing example

- Code for topic 9 provides versions of a program with different ways of debouncing buttons.
  - Example1, BounceDemo
  - Example2, DelayDebounceDemo
  - Example3, NonblockingDebounceDemo

Debouncing example BounceDemo

```c
#define F_CPU (16000000UL)

#include <avr/io.h>




void setup(void) {

}

void process(void) {


}

int main(void) {

    setup();

    for ( ;; ) {
        process();

    }
}
```

```c

void process(void) {

  //define a buffer to be sent
   unsigned char temp_buf[64];

   snprintf( (char *)temp_buf, sizeof(temp_buf), "%d\n", counter );


 //detect pressed switch on D7
 if (BIT_IS_SET(PIND,7)){

   while ( BIT_IS_SET(PIND, 7) ) {
     // Block until switch released.
   }

  //increment count
   counter++;

 }

   //detect presed switch on D6
 if (BIT_IS_SET(PIND,6)){

  //send serial data
   uart_putstring(temp_buf);

   while ( BIT_IS_SET(PIND, 6) ) {
       // Block until switch released.
       }

   }

  //flash LED every time increments of 5 occur
   if(counter%5 == 0)
     PORTB^=(1<<PINB5);

```

Debouncing example
Delay-based de-bouncing

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



#define DEBOUNCE_MS (150)

void setup(void) {
}

void process(void) {
}

int main(void) {
        setup();

        for ( ;; ) {
                process();
        }
}
```

```

void process(void) {

   //define a buffer to be sent
    unsigned char temp_buf[64];

    snprintf( (char *)temp_buf, sizeof(temp_buf), "%d\n", counter );


  //detect pressed switch on D7
  if (BIT_IS_SET(PIND,7)){

   _delay_ms(DEBOUNCE_MS);

    while ( BIT_IS_SET(PIND, 7) ) {
            // Block until switch released.
    }

   //increment counter
    counter++;

  }

    //detect presed switch on D6
  if (BIT_IS_SET(PIND,6)){

   //send serial data
    uart_putstring(temp_buf);

    while ( BIT_IS_SET(PIND, 6) ) {
            // Block until switch released.
        }

    }

   //flash LED every time increments of 5 occur
    if(counter%5 == 0)
      PORTB^=(1<<PINB5);


}

```

real worl

Debouncing example

king de-bouncing

```c
//threshold used for debouncing
#define THRESHOLD (1000)
```

```c
void process(void) {

   //define a buffer sent
    unsigned char temp_buf[64];

    snprintf( (char *)temp_buf, sizeof(temp_buf), "%d\n", counter );


  //detect pressed switch on D7
  if (left_button_clicked() ){

   //increment count
    counter++;

  }

    //detect pressed switch on D6
  if (BIT_IS_SET(PIND,6)){

   //send serial data
    uart_putstring(temp_buf);

    while ( BIT_IS_SET(PIND, 6) ) {
    // Block until switch released.
    }

  }

   //flash LED every time increments of 5 occur
    if(counter%5 == 0)
    PORTB^=(1<<PINB5);


}
```

```c
bool left_button_clicked(void){

  bool was_pressed = pressed;

  if ( BIT_IS_SET(PIND, 7) ) {
        closed_num++;
        open_num = 0;

        if ( closed_num  > THRESHOLD ) {
            if ( !pressed ) {
                closed_num = 0;
            }

            pressed = true;
        }
    }
    else {
        open_num++;
        closed_num = 0;

        if ( open_num > THRESHOLD ) {
            if ( pressed ) {
                open_num = 0;
            }

            pressed = false;
        }
    }

    return was_pressed && !pressed;


}
```

Debouncing

- We have demonstrated switch bounce, and examined ways to address the problem
- A non-blocking algorithm has been developed which is very good, but still relies on polling.
- To perfect the algorithm, we need a way to sample the physical state of the switch at a fairly high and constant frequency.

Timers

- Timers are commonly used (or work by) to increment a counter variable (a register).
- They have a myriad of uses ranging from simple delay intervals right up to complex PWM generation.
- Timers are at the heart of automation.

Timers

- The AVR timers are very useful as they can run asynchronous to the main AVR core.

- This is a fancy way of saying that the timers are separate circuits on the AVR chip which can run independently of the main program, interacting via the control and count registers, and the timer interrupts.

- Timers can be configured to produce outputs directly to pre-determined pins, reducing the processing load on the AVR core.

Timers

- Like all digital systems, the timer requires a clock in order to function.
- As each clock pulse increments the timer's counter by one, the timer measures intervals in periods of one on the input frequency.
- This means the smallest amount of time the timer can measure is one period of the incoming clock signal.
- The clock source (from the internal clock or an external source) sends pulses to the prescaler which divides the pulses by a determined amount.
- This input is sent to the control circuit which increments the TCNTn register. When the register hits its TOP value it resets to 0 and sends a TOVn (timer overflow) signal which could be used to trigger an interrupt.

Timers

- Our microcontroller has four timers, Timer 0, Timer 1 and Timer 2
- Each timer is associated to a counter and a clock signal.
- The counter is incremented by 1 in every period of the timer's clock signal
- The clock signal can come from
  - The internal system clock
  - An external clock signal

Timers

- Timers store their values into internal 8 or 16 bit registers, depending on the size of the timer being used.

- These registers can only store a finite number of values, resulting in the need to manage the timer (via prescaling, software extension, etc) so that the interval to be measured fits within the range of the chosen timer.

Timers

- What happens when the range of the timer is exceeded.
  - Does the AVR explode? Does the application crash? Does the timer automatically stop?
- In the event of the timer register exceeding its capacity, it will automatically roll around back to zero and keep counting.
- When this occurs, we say that the timer has "overflowed".

Timers: Overflow

- When an overflow occurs, a bit is set in one of the timer status registers to indicate to the main application that the event has occurred.
- There is also a corresponding bit which can enable an interrupt to be fired up each time the timer resets back to zero.

Timers
How often does the timer overflow?

- Clock speed (16MHz)
- Prescaler (1, 8, 64, 256, 1024)
- Counter size (8 or 16 bit)

- Timer 0: Normal timer mode, Prescaler of 1024, 8 Bit timer
- Timer Speed
  = 1 / (Clock Speed/Prescaler)
  = 1 / (16000000/1024)
  = 64 micro seconds (sometime also called timer resolution)

- Timer Overflow Speed
  = (Timer Speed * 256)
  = 16384 micro seconds

Timers

- Three timers (the atmega328)
  - Timer 0, is a 8 bits
  - Timer 1, is a 16 bits
  - Timer 2, is a 8 bit
- Let's focus on Timer 0, its registers are
  - TCCR0A
  - TCCR0B ← Setting timer pre-scaler
  - TCNT0 ← Where the count is stored
  - OCR0A
  - OCR0B
  - TIMSK0
  - TIFR0
- They can be used set the behavior of the timer, override direction of I/O pins, configure clock, pre-scaler, and also to set specific triggers to start the timer.

Timer Registers: TCCR0B setting timer pre-scaler

Normal operation=0

13.8.2 Timer/Counter Control Register B – TCCR0B

Bit	7	6	5	4	3	2	1	0	
	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

prescaler

- **Bits 2:0 – CS02:0: Clock Select**

The three Clock Select bits select the clock source to be used by the Timer/Counter.

Table 13-9. Clock Select Bit Description

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.

Timers
setting a timer

```
#define FREQ (16000000.0)
#define PRESCALER (1024.0)

void setup(void) {


        CLEAR_BIT(TCCR0B,WGM02);   0
        SET_BIT(TCCR0B,CS02);      1
        CLEAR_BIT(TCCR0B,CS01);    0  } 1024
        SET_BIT(TCCR0B,CS00);      1
}

void process(void) {
        // The value of the counter is a number of ticks
        // To convert from ticks back to seconds,
        // we multiply by the pre-scaler and divide by clock speed.
        double time = TCNT0 * PRESCALE / FREQ;
}

int main(void) {
        setup();

        for ( ;; ) {
                process();
        }
}
```

Definition: Frequency = 1 / Period.

Figures in this table assume that the CPU speed is set to 16MHz in the **setup** *phase.*

cs02:0	Pre-scaler	Counter frequency	Overflow period = 256/freq	Overflow frequency
0b000	0	0	n/a	n/a
0b001	1	16MHz	0.000016s	62.5kHz
0b010	8	2MHz	0.000128s	7.8125kHz
0b011	64	250kHz	0.001024s	976.56Hz
0b100	256	62.500kHz	0.004096s	244.14Hz
0b101	1024	15.625kHz	0.016384s	61.035Hz

Normal operation=0

13.8.2 Timer/Counter Control Register B – TCCR0B

Bit	7	6	5	4	3	2	1	0	
	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

prescaler

- **Bits 2:0 – CS02:0: Clock Select**

The three Clock Select bits select the clock source to be used by the Timer/Counter.

Interrupts

- In most microcontrollers, there is a function called interrupt. This interrupt can be fired up whenever certain condition is 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.

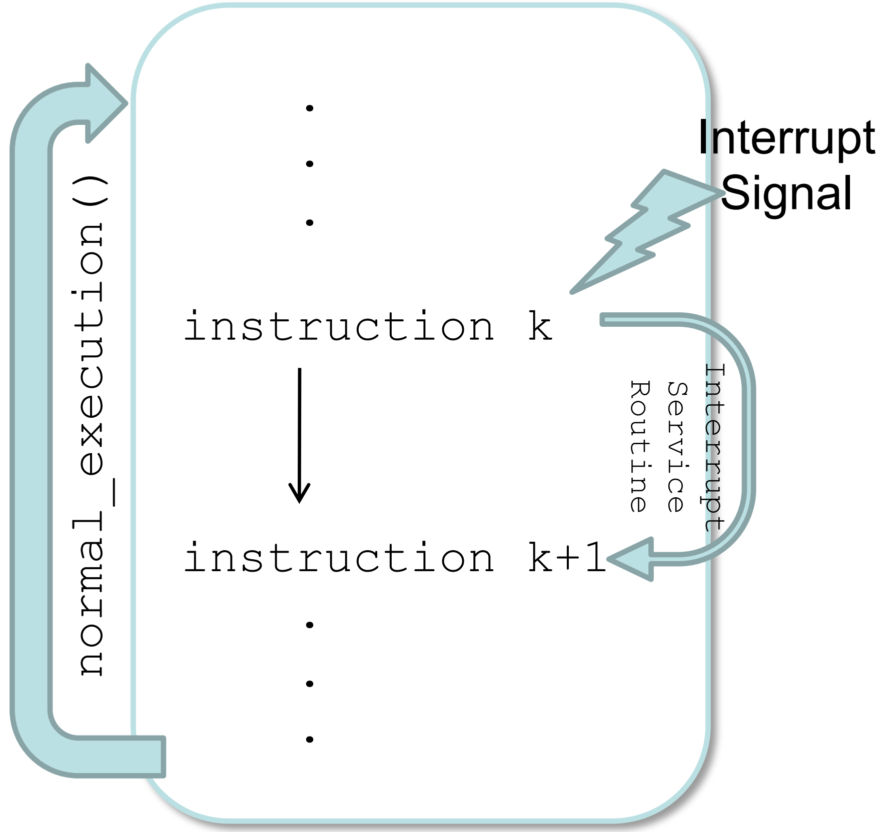
Polling

- In all of the solutions to the pracs throughout this semester, the methods you've have been using are what it is known as polling.
- A polling method is one which constantly loops over the same code that checks each iteration to see if a particular state has changed (i.e. performing a desired action by continuously checking if a button has been pressed).
- In other words, when we are polling we are waiting for a specific event. This action is considered blocking due to the fact that no other code can be run until this condition is met.
- Consequently, this blocking code can cause problems in larger embedded systems where you have multiple inputs and outputs. It is in scenarios like these where the distinct advantage of interrupts is apparent.

Polling vs. Interrupts

```
while (1)
{
    check_device_status();
    if(service_required)
    {
        service_routine();
    }

    normal_execution();
}
```



Interrupts

- Interrupts are a crucial part of writing code for embedded systems. In performing the interrupt, the microcontroller goes through the following three steps:
  1. Halts the current process (noting where it is in this process)
  2. Runs the interrupt code until it completes
  3. Returns back to the original process and continue from where it was before the interrupt

Interrupts

- The interrupt code is called the interrupt service routine (ISR). The syntax for code that creates an ISR is similar to that of a function (except it does not need a declaration – only an implementation). An example of an ISR is shown below for the USART receive complete interrupt event (you must include avr/interrupt.h to run any interrupt related code):

```
ISR(USART_RX_vect) {
        // Code to be executed within the interrupt routine
}
```

- You can think of the ISR as basically an isolated section of code which will get called anytime an event occurs.

- As already discussed, there are many different types of interrupts. Each type has as its own interrupt vector. In the above example, the interrupt vector is the USART_RX_vect part. It is this part of the ISR declaration code that would need to be changed for a different interrupt event.

Interrupts

- There are three important conditions that must be met for the ISR to be called and executed correctly:
  1. The enable bit for global interrupts must be set. This allows the microcontroller to process interrupts via ISRs when set, and prevents them from running when cleared. It defaults to being cleared on power up, so we need to set it by using the sei() utility function. Conversely, you can clear it by using the cli() utility function.
  2. The individual interrupt source's enable bit must explicitly be set. Each hardware interrupt source has its own separate interrupt enable bit (this resides in the related peripheral's control registers).
  3. The condition for the interrupt must be met. For example, a character must have been received through USART for the USART receive complete (USART_RX) interrupt to be executed.

12.4 Interrupt Vectors in ATmega328 and ATmega328P

Table 12-6. Reset and Interrupt Vectors in ATmega328 and ATmega328P

VectorNo.	Program Address[2]	Source	Interrupt Definition
1	0x0000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2 OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1 OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow
18	0x0022	SPI, STC	SPI Serial Transfer Complete
19	0x0024	USART, RX	USART Rx Complete
20	0x0026	USART, UDRE	USART, Data Register Empty
21	0x0028	USART, TX	USART, Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE READY	EEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator
25	0x0030	TWI	2-wire Serial Interface
26	0x0032	SPM READY	Store Program Memory Ready

Notes: 1. When the BOOTRST Fuse is programmed, the device will jump to the Boot Loader address at reset, see "Boot Loader Support – Read-While-Write Self-Programming" on page 272.

1 reset interrupt

external interrupts

Pin Change interrupt

timer interrupts

serial port interrupts

Interrupts
Problem
Trigger an interrupt when a timer overflows

Steps:

1. Set up the timer with correct prescaler.
2. Turn on interrupts
3. Write the Interrupt Service Routine that is called when the timer overflows. Data shared between the ISR and your main program **must be both volatile and global in scope** in the C language. i. e

```
volatile int overflow_count;
```

Set up timer 0 overflow interrupt

6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2 OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1 OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow
18	0x0022	SPI, STC	SPI Serial Transfer Complete
19	0x0024	USART, RX	USART Rx Complete
20	0x0026	USART, UDRE	USART, Data Register Empty
21	0x0028	USART, TX	USART, Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE READY	EEPROM Ready

Set up Overflow Interrupt Enable

Timer/Counter Interrupt Mask Register – TIMSK0

Bit	7	6	5	4	3	2	1	0	
	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0	TIMSK0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bits 7..3, 0 – Res: Reserved Bits**

These bits are reserved bits and will always read as zero.

```
TIMSK0 |= (1 << TOIE0);
```

- **Bit 2 – OCIE0B: Timer/Counter Output Compare Match B Interrupt Enable**

When the OCIE0B bit is written to one, and the I-bit in the Status Register is set, the Timer/Counter Compare Match B interrupt is enabled. The corresponding interrupt is executed if a Compare Match in Timer/Counter occurs, i.e., when the OCF0B bit is set in the Timer/Counter Interrupt Flag Register – TIFR0.

- **Bit 1 – OCIE0A: Timer/Counter0 Output Compare Match A Interrupt Enable**

When the OCIE0A bit is written to one, and the I-bit in the Status Register is set, the Timer/Counter0 Compare Match A interrupt is enabled. The corresponding interrupt is executed if a Compare Match in Timer/Counter0 occurs, i.e., when the OCF0A bit is set in the Timer/Counter 0 Interrupt Flag Register – TIFR0.

- **Bit 0 – TOIE0: Timer/Counter0 Overflow Interrupt Enable**

When the TOIE0 bit is written to one, and the I-bit in the Status Register is set, the Timer/Counter0 Overflow interrupt is enabled. The corresponding interrupt is executed if an overflow in Timer/Counter0 occurs, i.e., when the TOV0 bit is set in the Timer/Counter 0 Interrupt Flag Register – TIFR0.

Set up timer 0 and overflow interrupt

```
void setup(void) {
        CLEAR_BIT(TCCR0B,WGM02);
        SET_BIT(TCCR0B,CS02);
        CLEAR_BIT(TCCR0B,CS01);
        SET_BIT(TCCR0B,CS00);

        // Enables the Timer Overflow interrupt for Timer 0
        SET_BIT(TIMSK0, TOIE0);

        // Enable global interrupt
        sei();
}

volatile int overflow_counter = 0;

ISR(TIMER0_OVF_vect) {
        overflow_counter ++;
}

void process(void) {
        double time = ( overflow_counter * 256.0 + TCNT0 ) * PRESCALE  / FREQ;
}

int main(void) {
        setup();

        for ( ;; ) {
                process();
        }
}
```

Summary

- Debouncing
  - Quite important when working with buttons
  - Can be implemented using interrupts
- Timers and Interrupts
  - Needed to generate events or execute part of your program with good time predictability.
- Example code on Blackboard