Topic 9:
Debouncing, Timers and Interrupts

Luis Mejias

Outline

- Review Topic 8: LCD Screen
- Debouncing
- Timers
- Interrupts
- Examples

Review Topic 8: LCD screen

- There are primarily two ways to write pixel data to the LCD
  - CAB202_teensy library
  - Direct screen write to the LCD

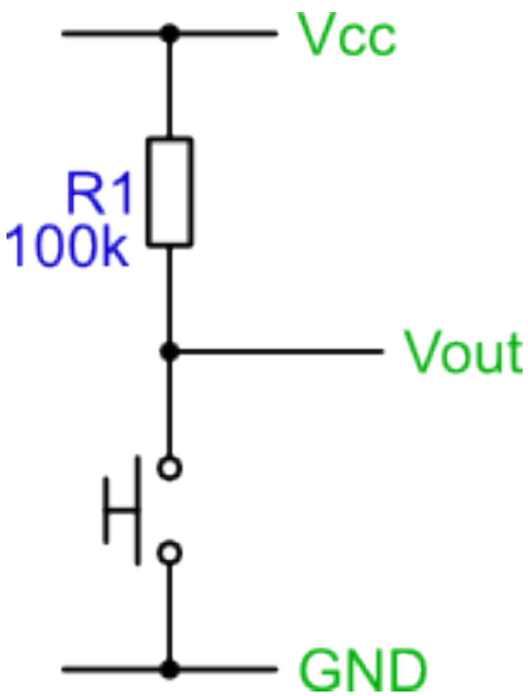
- CAB202_teensy library provides a high level interface (functions) to write characters on the LCD screen.

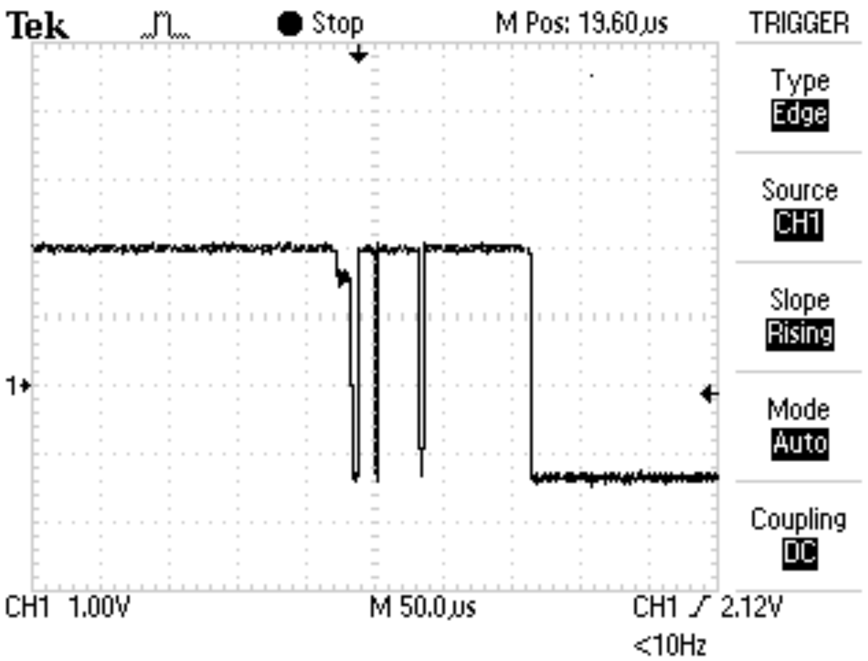
- lcd_model.h provides helpful macros to use for the direct screen writing to the LCD

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

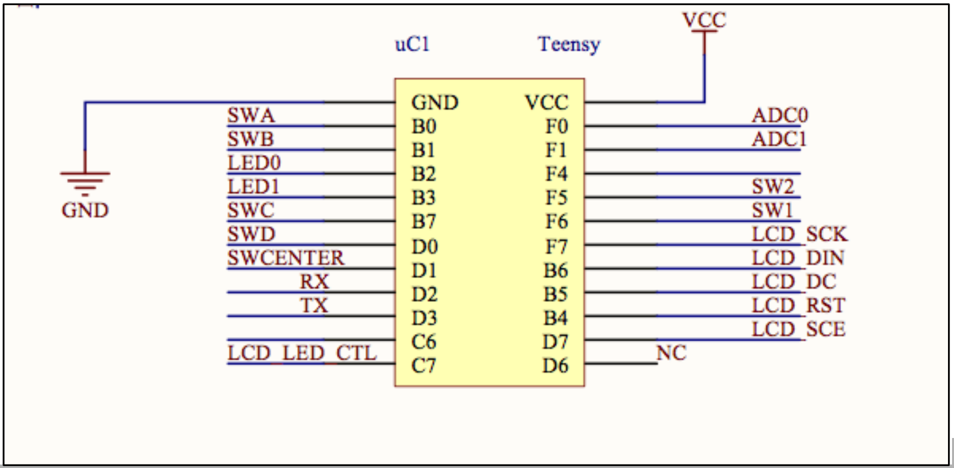
- Code for topic 9 provides versions of a program with different ways of debouncing buttons.
  - BounceDemo
  - DelayDebounceDemo
  - NonblockingDebounceDemo

Debouncing example BounceDemo

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

#include <graphics.h>
#include <macros.h>

void setup(void) {

}

void process(void) {
}

int main(void) {
}
```

```
int main(void) {
        setup();

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

```
void setup(void) {
        CLEAR_BIT(DDRD, 1); // up
        CLEAR_BIT(DDRF, 6); // left_button
}
```



Debouncing example BounceDemo

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

#include <graphics.h>
#include <macros.h>

void setup(void) {

}

void process(void) {
}

int main(void) {
}
```

```
uint16_t counter = 0;

void process(void) {
		// Detect a Click on left button
		if ( BIT_IS_SET(PINF, 6) ) {
			while ( BIT_IS_SET(PINF, 6) ) {
				// Block until button released.
			}
			// Button has now been pressed and released...
			counter ++;
		}

		// Display and wait if joystick up.
		if ( BIT_IS_SET(PIND, 1) ) {
			draw_all();

			while ( BIT_IS_SET(PIND, 1) ) {
				// Block until joystick released.
			}
		}
}
```

Debouncing example
Delay-based de-bouncing

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

#include <graphics.h>
#include <macros.h>
#include <lcd_model.h>

#define DEBOUNCE_MS (150)

void setup(void) {
}

void process(void) {
}

int main(void) {
	setup();

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

```
void setup(void) {
	CLEAR_BIT(DDRD, 1); // up
	CLEAR_BIT(DDRF, 6); // left_button
}
```

```
void process(void) {
	// Detect a Click on left button
	if ( BIT_IS_SET(PINF, 6) ) {
		_delay_ms(DEBOUNCE_MS);

		while ( BIT_IS_SET(PINF, 6) ) {
			// Block until button
released.
		}
		// Button has now been pressed and
released...
		counter ++;
	}
	// Display and wait if joystick up.
	if ( BIT_IS_SET(PIND, 1) ) {
		draw_all();
		while ( BIT_IS_SET(PIND, 1) ) {
			// Block until joystick
released.
		}
	}
}
```

real worl

Debouncing example
non-blocking de-bouncing

```c
void setup(void) {
set_clock_speed(CPU_8MHz);
lcd_init(LCD_DEFAULT_CONTRAST);
draw_all();
CLEAR_BIT(DDRD, 1); // up
CLEAR_BIT(DDRF, 6); // left_button
}

uint16_t counter = 0;
void process(void) {
if ( left_button_clicked() ) {
        counter++;
}
// Display and wait if joystick up.
if ( BIT_IS_SET(PIND, 1) ) {
   draw_all();
   while ( BIT_IS_SET(PIND, 1) ) {
        // Block until joystick released.
    }
  }
}

int main(void) {
setup();
for ( ;; ) {
   process();
  }
}
```

```c
#define THRESHOLD (1000)
bool pressed = false;
uint16_t closed_num = 0;
uint16_t open_num = 0;
bool left_button_clicked(void) {
        // Detect a Click on left button
        bool was_pressed = pressed;

        if ( BIT_IS_SET(PINF, 6) ) {
                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 a two 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, Timer 3, and Timer 4
- 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 (8MHz)
- 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 / (8000000/1024)
  = 128 micro seconds (sometime also called timer resolution)

- Timer Overflow Speed
  = (Timer Speed * 256)
  = 32768 micro seconds

Timers

- Four timers (the atmega32U4 doesn't have a timer 2)
  - Timers 1 and 3 are 16 bits
  - Timer 0, is a 8 bits
  - Timer 4, is a 10 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      (8000000.0)
#define PRESCALE (1024.0)

void setup(void) {
        set_clock_speed(CPU_8MHz);

        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 8MHz 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	8MHz	0.000032s	31.25kHz
`0b010`	8	1MHz	0.000256s	3.90625kHz
`0b011`	64	125kHz	0.002048s	488.28125Hz
`0b100`	256	31.25kHz	0.008192s	122.0703125Hz
`0b101`	1024	7.8125kHz	0.032768s	30.517578125Hz

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.

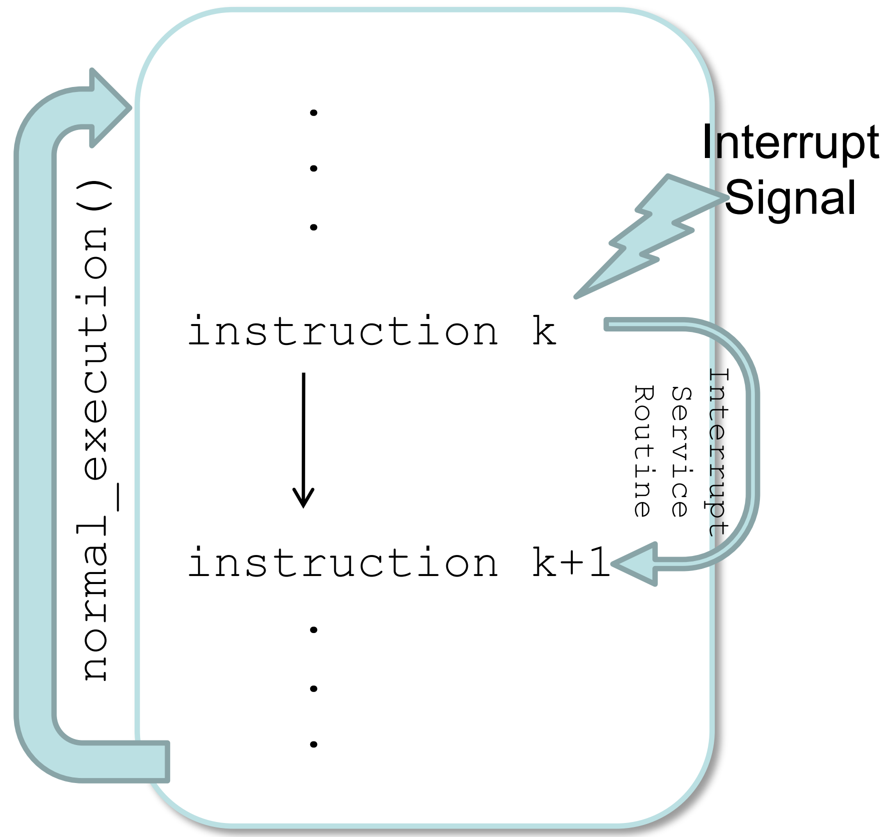
- A polling method is one which constantly loops over the same code that checks each iteration to see if a particular state has changed (i.e. performing a desired action by continuously checking if a button has been pressed).

- 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_RXC_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_RXC_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_RXC) interrupt to be executed.

9.1 Interrupt Vectors in ATmega16U4/ATmega32U4

Table 9-1. Reset and Interrupt Vectors

Vector No.	Program Address[2]	Source	Interrupt Definition
1	$0000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	INT3	External Interrupt Request 3
6	$000A	Reserved	Reserved
7	$000C	Reserved	Reserved
8	$000E	INT6	External Interrupt Request 6
9	$0010	Reserved	Reserved
10	$0012	PCINT0	Pin Change Interrupt Request 0
11	$0014	USB General	USB General Interrupt request
12	$0016	USB Endpoint	USB Endpoint Interrupt request
13	$0018	WDT	Watchdog Time-out Interrupt
14	$001A	Reserved	Reserved
15	$001C	Reserved	Reserved
16	$001E	Reserved	Reserved
17	$0020	TIMER1 CAPT	Timer/Counter1 Capture Event
18	$0022	TIMER1 COMPA	Timer/Counter1 Compare Match A
19	$0024	TIMER1 COMPB	Timer/Counter1 Compare Match B
20	$0026	TIMER1 COMPC	Timer/Counter1 Compare Match C
21	$0028	TIMER1 OVF	Timer/Counter1 Overflow
22	$002A	TIMER0 COMPA	Timer/Counter0 Compare Match A
23	$002C	TIMER0 COMPB	Timer/Counter0 Compare match B
24	$002E	TIMER0 OVF	Timer/Counter0 Overflow
25	$0030	SPI (STC)	SPI Serial Transfer Complete
26	$0032	USART1 RX	USART1 Rx Complete
27	$0034	USART1 UDRE	USART1 Data Register Empty
28	$0036	USART1TX	USART1 Tx Complete
29	$0038	ANALOG COMP	Analog Comparator

1 reset interrupt

5 external interrupts

1 Pin Change interrupt

8 timer interrupts

serial port interrupts

. 000213J

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

9	$0010	Reserved	Reserved
10	$0012	PCINT0	Pin Change Interrupt Request 0
11	$0014	USB General	USB General Interrupt request
12	$0016	USB Endpoint	USB Endpoint Interrupt request
13	$0018	WDT	Watchdog Time-out Interrupt
14	$001A	Reserved	Reserved
15	$001C	Reserved	Reserved
16	$001E	Reserved	Reserved
17	$0020	TIMER1 CAPT	Timer/Counter1 Capture Event
18	$0022	TIMER1 COMPA	Timer/Counter1 Compare Match A
19	$0024	TIMER1 COMPB	Timer/Counter1 Compare Match B
20	$0026	TIMER1 COMPC	Timer/Counter1 Compare Match C
21	$0028	TIMER1 OVF	Timer/Counter1 Overflow
22	$002A	TIMER0 COMPA	Timer/Counter0 Compare Match A
23	$002C	TIMER0 COMPB	Timer/Counter0 Compare match B
24	$002E	TIMER0 OVF	Timer/Counter0 Overflow
25	$0030	SPI (STC)	SPI Serial Transfer Complete
26	$0032	USART1 RX	USART1 Rx Complete
27	$0034	USART1 UDRE	USART1 Data Register Empty
28	$0036	USART1TX	USART1 Tx Complete
29	$0038	ANALOG COMP	Analog Comparator

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	

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

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

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

- **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

```
#define FREQ     (8000000.0)
#define PRESCALE (1024.0)

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