Laboratory 4 – Usage of timers

1. Timer based interrupts

Beside external interrupt, the MCU responds to internal ones which are triggered by external events (on the external pins). The source of the internal interrupts are hardware components inside the MCU (like the Timers).

Using the timer/counter interrupts, you can trigger events at specific time intervals (without using functions like *delay* and *millis*). Because these interrupts have an asynchronous behavior, the main program flow can be executed normally and only when a temporal threshold is reached a specific ISR is executed. The timer increments a *counter* register, and when its maximum counting capacity is reached an *overflow* bit (*flag*) is set. The flag can be manually checked or it can trigger an interrupt request. At the ISR execution the fag is reset.

Each timer needs a clock source, which most commonly is the development board's oscillator; its frequency can be divided by an auxiliary counter (*prescaler*).

Arduino Mega has 5 counters (2 / 8 bits and 3 / 16bits). The Arduino UNO has 2 / 8 bits and 1 / 16bits counters. To use the counters the configuration registers must be properly set. Two of them are the Timer Counter Control Registers TCCRxA and TCCRxB (x denotes the timer number). To start the timer the most important bits are those that set the clock frequency division (*prescaler*) (and consequently the counting speed): CSn2, CSn1 and CSn0.

TCCR1A – Timer/Counter 1 Control Register A

Bit	7	6	5	4	3	2	1	0	
(0x80)	COM1A1	COM1A0	COM1B1	COM1B0	COM1C1	COM1C0	WGM11	WGM10	TCCR1A
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	

TCCR1B – Timer/Counter 1 Control Register B

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Clock Select Bit Description

CS_X2	CS_X1	CSx0	Description
0	0	0	No clock source. (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/1 (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 T_X pin. Clock on falling edge
1	1	1	External clock source on T_X pin. Clock on rising edge

By default CSx2, CSx1 and CSx0 are 0, meaning that the timer does not have a clock signal and therefore is stopped. A valid configuration for the prescaler means starting the timer. If the selected clock source is external, the timer will work only if such an external source is connected to the board (which is not the case at the lab – such settings will be avoided!!!).

Below is a summary of the most important registers used with the timers:

- TCCRx - Timer/Counter Control Register: here you can set the clock source.
- TCNTx - Timer/Counter Register: stores the current value of the timer's counter.
- OCRx - Output Compare Register: value set by the user and compared with the TCNTx to generate waveforms or trigger events.
- ICRx - Input Capture Register: used to measure time intervals between external events (only for 16 bit timers).
- TIMSKx - Timer/Counter Interrupt Mask Register: to enable/disable timer interrupts.
- TIFRx - Timer/Counter Interrupt Flag Register: signals the existence of an interrupt request.

In the following example we will increment a variable when TIMER1 overflows. The value of the incremented variable will be displayed on the LCD.

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

LiquidCrystal lcd(7, 6, 5, 4, 3, 2);
volatile int myVar;

void setup()
{
 myVar = 0;
 //init Timer1
 cli(); // disable the global interrupts system in order to setup the timers
 TCCR1A = 0; // SET TCCR1A and B to 0
 TCCR1B = 0; // timer set to Normal mode (WGMx3:0 = 0)
 TIMSK1 = (1 << TOIE1); //timer overflow interrupt enable for timer 1
 //Set the prescaler to 1024
 // CPU freq. is 16 MHZ and Timer1 is on 16 bits
 // Counter increment at every dt = 1024 / (16 * 10^6) sec
 // Overflow event occurs at every: dt * 2^16 = 4.194 sec.
 TCCR1B |= (1 << CS10);
 TCCR1B |= (1 << CS12);
 lcd.begin(16, 2);
 lcd.print("Timers");
 sei(); // enable the global interrupts system
 }

void loop()
{
 lcd.setCursor(0,1);
 lcd.print(myVar);
 lcd.setCursor(5, 1);
 lcd.print(TCNT1);
}

ISR(TIMER1_OVF_vect)
{
  myVar = myVar + 1;
}
```

In order to trigger the timer interrupt at customized time intervals (not only on timer overflow), the CTC (Clear Timer on Compare match) mode must be used. This way, the value of the TCNTx register will be compared with the value of the OCRx register (set by the user) and at equality the value of TCNTx is reset (becomes zero).

In order to obtain a T time interval between consecutive interrupt requests, the value pf OCRx register must be set by the user:

$$OCRx + 1 = T / (P / (16 * 10^6)) \qquad (1)$$

Where:

T – the time interval between consecutive events (ex. 1 sec)
P – prescaler value (ex. 1024) (+ 1 is added to OCRx because the timer reset takes 1 clock cycle)

Applying eq. 1 for T=1 sec, we get OCRx =15624.

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

LiquidCrystal lcd(7, 6, 5, 4, 3, 2);
volatile int myVar;
void setup()
{
   // Initialize Timer1
   cli();   // disable the global interrupts system in order to setup the timers
   TCCR1A = 0; // SET TCCR1A and B to 0
   TCCR1B = 0;
   // Set the OCR1A with the desired value:
   OCR1A = 15624;
   // Active CTC mode:
   TCCR1B |= (1 << WGM12);
   // Set the prescaler to 1024:
   TCCR1B |= (1 << CS10);
   TCCR1B |= (1 << CS12);
   // Enable the Output Compare interrupt (by setting the mask bit)
   TIMSK1 |= (1 << OCIE1A);

   lcd.begin(16, 2);
   lcd.print("Timere cu CTC");

   sei(); // enable global interrupts
}

void loop()
{
 lcd.setCursor(0,1);
 lcd.print(myVar);
 lcd.setCursor(5, 1);
 lcd.print(TCNT1);
}
```

```
ISR(TIMER1_COMPA_vect)
{
   myVar = myVar + 1;
}
```

In contrast to the previous example, when the variable is incremented at overflow (about every 4 seconds), in the current example we have more control over the incrementing period of the variable (once per second).

In the following a much simpler method for using the timers, based on the TimerOne library, is presented. The TimerOne library can be downloaded from the following location: http://playground.arduino.cc/Code/Timer1, where detailed installation and usage instructions are given.

In the example below a routine is called every second:

```
#include <TimerOne.h>
#include <LiquidCrystal.h>
LiquidCrystal lcd(7, 6, 5, 4, 3, 2);

volatile int myVar;

void setup(void)
{
 Timer1.initialize(1000000);  // init the timing interval for event triggering (µs = 10^-6s)
 Timer1.attachInterrupt(ShowMessage);  // The function is called at the preset time interval
}

void ShowMessage(void)
{
 lcd.setCursor(0,0);
 lcd.print(myVar);
 myVar++;
}

void loop(void)
{ // Do something else …
}
```

2. Tones generation

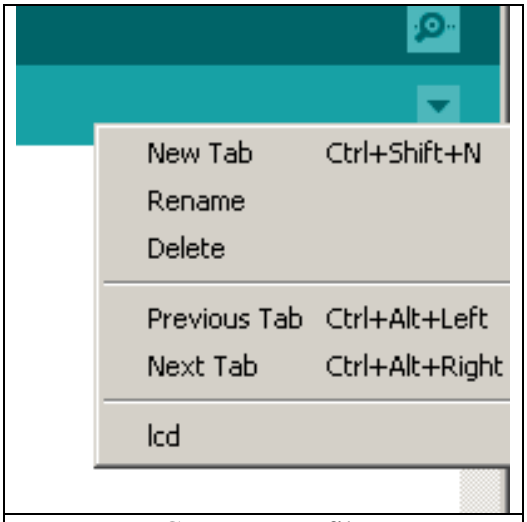
For that purpose Arduino offers the **tone** function, which generates specified frequency pulses with 50% fill factor. To call the function the length of the signal must be specified (otherwise the tone will be generated until the **noTone()** function is called. Minimum tone frequency is 31 HZ. In order to generate tones on different pins, **noTone()** must be called before switching the output pin.

The format of the tone function is:

tone (pin, frequency, duration)
or
tone (pin, frequency, duration)

In order to run the following example, the following connection must be made: connect **the red (signal) pin** of the speaker to the **digital pin 8 of the Arduino**. The **black pin** of the speaker must be connected to the ground (**GND**).

Afterwards, create a file named pitches.h. To create a new file inside a project, press the top-right button from the IDE and select the "New Tab" option (see figure bellow):



Create a new file

Rename the new file *pitches.h* and insert the tones defined in appendix A. Save the document and go back to the principal tab, where enter the following code:

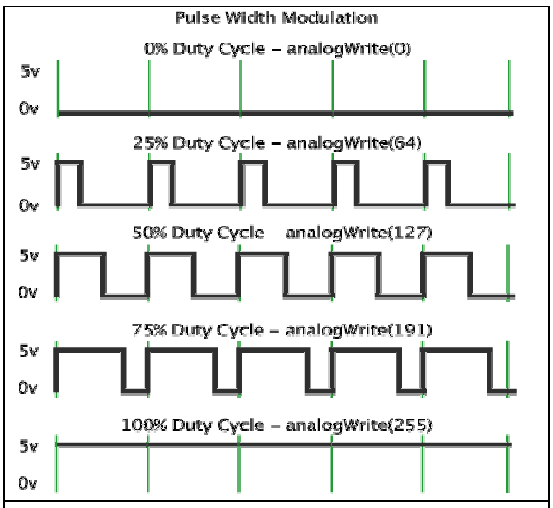
```
#include "pitches.h“ // contains the frequency values for all the notes

// notes in the melody – constant values defining frequency for each used note
int melody[] =
  {NOTE_C4,  NOTE_G3,NOTE_G3,  NOTE_A3,  NOTE_G3,0,  NOTE_B3,  NOTE_C4};

// note durations: 4 = quarter note, 8 = eighth note, etc.:
int noteDurations[] = { 4,8,8,4,4,4,4,4 };

void setup()
 {
  for (int thisNote = 0; thisNote < 8; thisNote++)
   { // iterate over the notes of the melody:
    // to calculate the note duration, take one second divided by the note type.
    // e.g. quarter note = 1000 / 4, eighth note = 1000/8, etc.
    int noteDuration = 1000/noteDurations[thisNote];
    tone(8, melody[thisNote],noteDuration);
   // to distinguish the notes, set a minimum time between them: note's duration + 30%
    int pauseBetweenNotes = noteDuration * 1.30;
   delay(pauseBetweenNotes);
   noTone(8); // stop the tone playing for current note
  }
}

void loop() { }   // no need to repeat the melody
```

3. PWM signal generation using Arduino

PWM (Pulse Width Modulation) is a technique used to obtain an analog signal from a digital one by alternating periodically the output between "1" and "0". The fraction of time for which the signal is high ("1") is called duty cycle. With Arduino the PWM signal can be generated in 3 ways:

- using the PWM working modes of the AVR timers
- using the analogWrite function
- by varying in software the amount of time at which a pin's output is logic high

In the following, the **analogWrite**(duty-cycle) function will be used. The possible values for the parameter duty_cycle are between 0 .. 255 (0 means no signal is generated; 255 means a continuous signal of 5 V is generated). Any value X in between means a signal with a duty cycle of X/255. The figure bellow illustrates some examples:



PWM signals with several duty cycles

In the following example we will generate a PWM signal for the piezoelectric speaker and a PWM signal with the same duty cycle for the on-board LED. The signal pin of the piezoelectric speaker is (red wire) is connected to digital pin 8 and the black wire to the ground.

```
int buzerPin = 8; // pin for attaching the piezoelectric speaker
int puls = 0;    // duty cycle, initially 0
int step = 10;    // duty cycle increment step
int ledPin = 13; // on-board LED

void setup()  {
  // Pin  direction setup
  pinMode(buzerPin, OUTPUT);
  pinMode(ledPin, OUTPUT);
}

void loop()  {
  // set the duty cycle for both the buzzer and the LED
  analogWrite(buzerPin, puls);
  analogWrite(ledPin, puls);
  // increment the duty cycle
  puls = puls + step;
```

```
  // change the increment direction at the end of the interval
  if (puls <= 0 || puls >= 255) {
    step = -step ;
  }
  // small delay to sense the effects
  delay(30);
}
```

Individual work

1. Implement all the examples. Ask the teacher for any question related to theoretical or practical aspects.
2. Using the interrupts, play the melody in the background while the main program displays an animation on the LCD (ex. walking character). The animation speed should be variable to demonstrate the asynchronous playing of the melody relative the main program.
3. Using the button block, implement a mini-piano with 4 notes. Be sure to end the sound generation when the button is released.
4. Using the PWM technique and an LED block, implement an animation through which the intensity of each LED is varied continuously

References

1.Datasheet ATMega 2560 http://www.atmel.com/images/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf
2. Timer 1 library http://playground.arduino.cc/Code/Timer1
3. analogWrite functions https://www.arduino.cc/en/Reference/AnalogWrite
4. Tones generation: https://www.arduino.cc/en/Reference/Tone

Apendix: contents of pitches.h file

```
#define NOTE_B0  31
#define NOTE_C1  33
#define NOTE_CS1 35
#define NOTE_D1  37
#define NOTE_DS1 39
#define NOTE_E1  41
#define NOTE_F1  44
#define NOTE_FS1 46
#define NOTE_G1  49
#define NOTE_GS1 52
#define NOTE_A1  55
#define NOTE_AS1 58
#define NOTE_B1  62
#define NOTE_C2  65
#define NOTE_CS2 69
#define NOTE_D2  73
#define NOTE_DS2 78
#define NOTE_E2  82
#define NOTE_F2  87
#define NOTE_FS2 93
```

```
#define NOTE_G2  98
#define NOTE_GS2 104
#define NOTE_A2  110
#define NOTE_AS2 117
#define NOTE_B2  123
#define NOTE_C3  131
#define NOTE_CS3 139
#define NOTE_D3  147
#define NOTE_DS3 156
#define NOTE_E3  165
#define NOTE_F3  175
#define NOTE_FS3 185
#define NOTE_G3  196
#define NOTE_GS3 208
#define NOTE_A3  220
#define NOTE_AS3 233
#define NOTE_B3  247
#define NOTE_C4  262
#define NOTE_CS4 277
#define NOTE_D4  294
#define NOTE_DS4 311
#define NOTE_E4  330
#define NOTE_F4  349
#define NOTE_FS4 370
#define NOTE_G4  392
#define NOTE_GS4 415
#define NOTE_A4  440
#define NOTE_AS4 466
#define NOTE_B4  494
#define NOTE_C5  523
#define NOTE_CS5 554
#define NOTE_D5  587
#define NOTE_DS5 622
#define NOTE_E5  659
#define NOTE_F5  698
#define NOTE_FS5 740
#define NOTE_G5  784
#define NOTE_GS5 831
#define NOTE_A5  880
#define NOTE_AS5 932
#define NOTE_B5  988
#define NOTE_C6  1047
#define NOTE_CS6 1109
#define NOTE_D6  1175
#define NOTE_DS6 1245
#define NOTE_E6  1319
#define NOTE_F6  1397
#define NOTE_FS6 1480
#define NOTE_G6  1568
#define NOTE_GS6 1661
#define NOTE_A6  1760
#define NOTE_AS6 1865
```

```
#define NOTE_B6  1976
#define NOTE_C7  2093
#define NOTE_CS7 2217
#define NOTE_D7  2349
#define NOTE_DS7 2489
#define NOTE_E7  2637
#define NOTE_F7  2794
#define NOTE_FS7 2960
#define NOTE_G7  3136
#define NOTE_GS7 3322
#define NOTE_A7  3520
#define NOTE_AS7 3729
#define NOTE_B7  3951
#define NOTE_C8  4186
#define NOTE_CS8 4435
#define NOTE_D8  4699
#define NOTE_DS8 4978
```