MIC2-LAB-V - Peripheral programming

The MIC2 workshops are a sequel to the MIC1 workshops. Both workshops even use the same hardware: the ATmega328p Xplained mini board. Instead of using an IDE for rapid prototyping, such as the Arduino IDE, a more professional IDE is introduced: Microchip Studio. The major advantage of using Microchip Studio instead of the Arduino IDE, is that we can use the debugger that is available on the ATmega328p Xplained mini board.

The main purpose of this course is that you learn how registers are used to access the hardware peripherals of the microcontroller. We will therefore not make use of libraries like we did in the Arduino IDE. Such libraries, that directly access registers in the microcontroller, are also known as *drivers*. During this course, you will learn how to read drivers created by others and how to write drivers yourself.

You must prepare for a workshop by studying the code examples, reading the documentation and start with the assignment. At the start of each workshop, the lecturer will discuss the theory described in this study guide, emphasize the most important parts of the code examples and answer questions. During the remainder of the workshop, you'll work on the assignment. If the assignment is not finished during the workshop, it must be finished at home.

1 Topics

The following list summarizes the topics that will be discussed:

- Week 1. Basics
- Week 2. Digital
- Week 3. Analog
- Week 4. Timers/Counters
- Week 5. Communication
- Week 6. Sensors and Actuators
- Week 7. Project work

2 Hardware and software requirements

This section describes the required hardware and software for these workshops.

2.1 Hardware

The workshops require the following hardware, **which is identical to MIC1**, that can be purchased as a bundle from ARLE:

- ATmega328p Xplained mini board
- Micro-USB cable
- Female headers
  If you are not comfortable soldering these headers on the ATmega328p Xplained mini board, ask a fellow student or your lecturer for help.
- Additional components, such as LEDs and switches

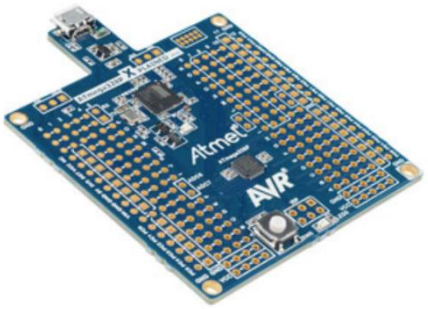


Image taken from https://www.microchip.com/DevelopmentTools/ProductDetails/ATMEGA328P-XMINI

The following hardware is available in the classroom:

- Resistors
- LEDs
- Potentiometers
- Switches
- Breadboard
- Jumper wires
- LCD
- Ultrasonic range sensor
- Servo motor

2.2 Software

Microchip Studio is used for software development. Download the installer for your operating system:

- https://www.microchip.com/en-us/development-tools-tools-and-software/microchip-studio-for-avr-and-sam-devices#
  Navigate to the Downloads section and select the "*Microchip Studio for AVR and SAM Devices - Web Installer*". When prompted during the installation, select the default options.

3 Pin mapping reference

The following overview can be used as a quick reference for the pin mappings.

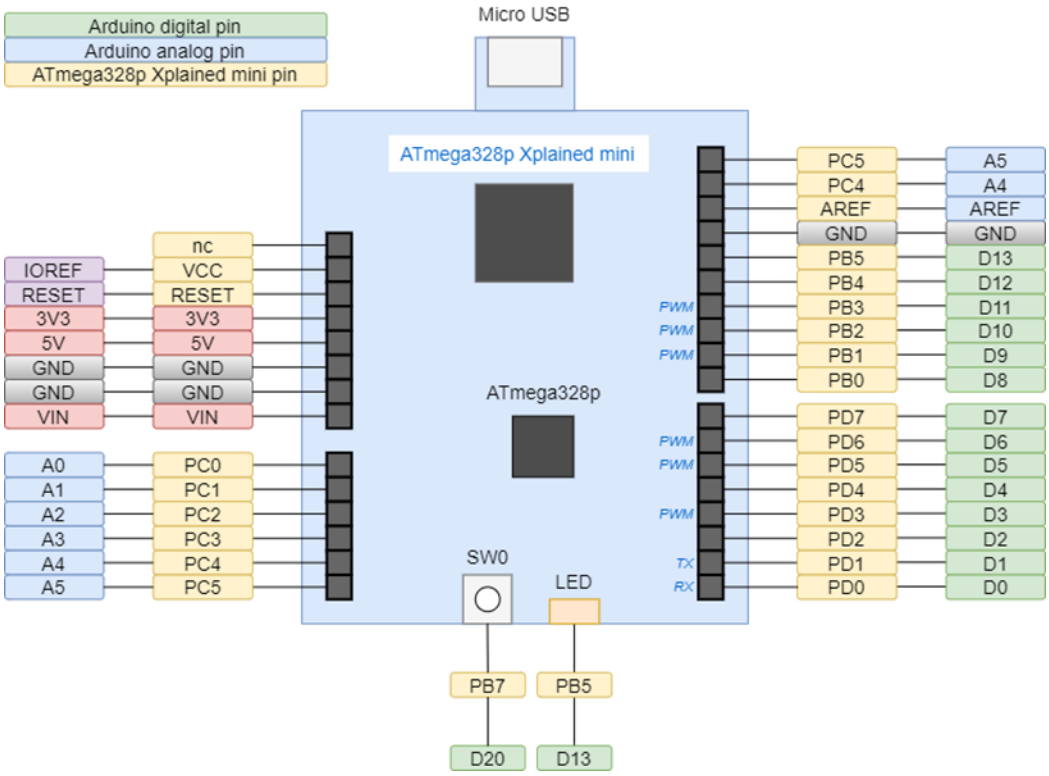


Image created by author.

4 Examination

The assignments are for practicing purposes. There is no need to hand them in. At the end of the semester, there will be a written multiple choice exam that covers all the topics from these classes.

5 Week 1. Basics

5.1 Introduction

Watch the following video:

> What is a microcontroller?
> https://youtu.be/jKT4H0bstH8

A microcontroller is used for creating embedded systems. It is a versatile component, because it can be reprogrammed, is very cheap, and consumes very little power. The building blocks of a simplified microcontroller are depicted as follows:

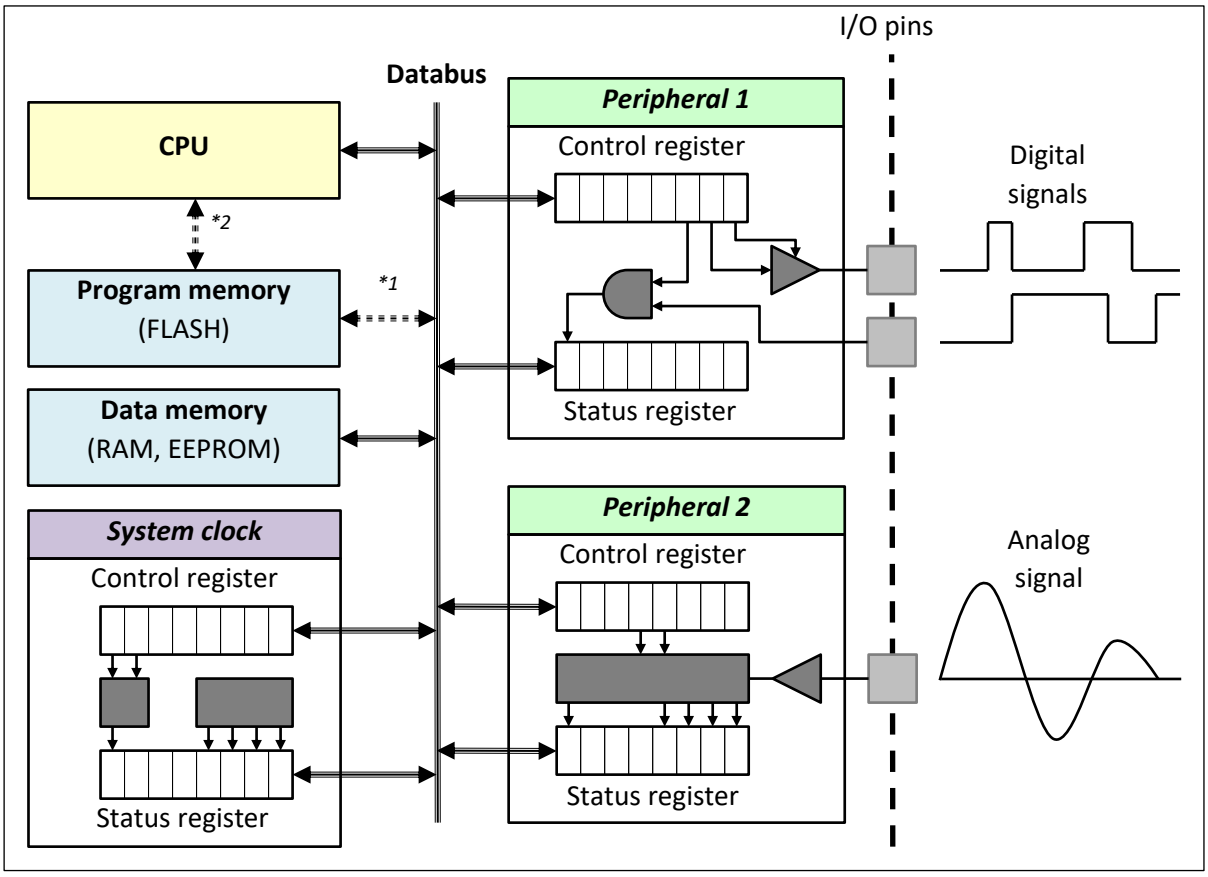


Image created by author.

The Central Processing Unit (**CPU**) executes machine instructions. These instruction are located in non-volatile **program memory**. These instructions are created by a software engineer and are also known as the application code. In case the program- and data memory use the same databus, we speak of a Von Neumann architecture (*1). In case the program- and data memory use a separate databus, we speak of a Harvard architecture (*2).

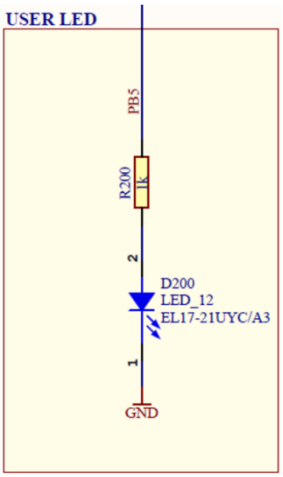
The machine instructions perform operations on **memory** and **peripherals**. A peripheral is on-chip hardware that provides additional functionality, such as digital I/O, analog I/O, timers, serial communication, etc. The block diagram shows two (fictional) peripherals. A peripheral is connected to external hardware by one or more I/O pins.

Controlling a peripherals hardware is achieved by writing data to a peripheral's **control registers**. These registers are also known as the programmer's interface to the hardware. The block diagram shows 8-bit registers, but there also exists microcontrollers with 16 or 32-bit registers. Checking the status of a peripheral is achieved by reading from **status registers**. The bits in the status registers are often referred to as flags. The meaning of the bits in the control and status registers is described by the vendor in a document called the **datasheet**. The **system clock** can be thought of as a peripheral that controls the speed at which the microcontroller is operating.

Reading from and writing to registers is achieved in the software application code. Here are two examples in the C programming language for accessing registers, assuming a peripheral has a control register defined as CTRL_REG and a status register defined as STAT_REG.

```
CTRL_REG = 15;                  // Assign the decimal value 15 to the register.
while(STAT_REG & (1<<BIT3)){;}  // Wait while bit 3 in the register is logic 1.
```

5.2 Hello World!

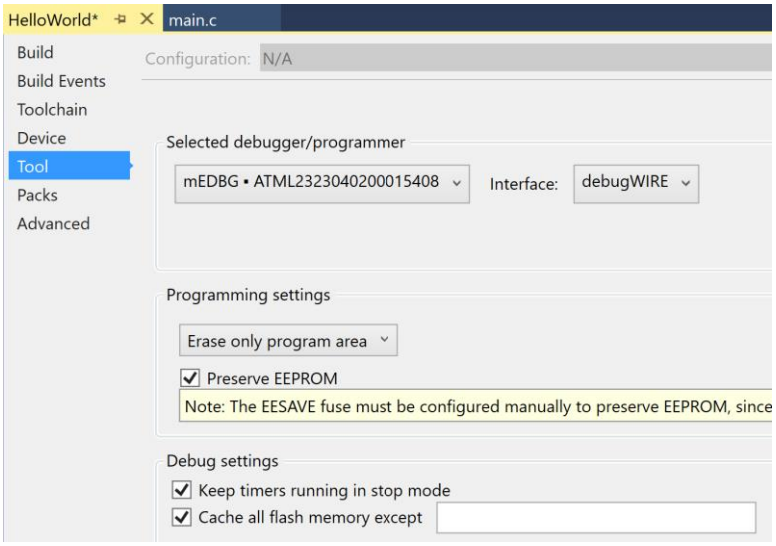


After having installed the IDE, it is common practice to make sure that everything was installed and configured correctly by creating a "*Hello World!*" application. In microcontroller programming, this means making an LED blink at a known frequency (often approximately 1 Hz). The ATmega328p Xplained mini board has an LED connected to pin PB5, as shown in the adjacent image taken from the board's user guide (User Guide, 2017). This LED will be used in the following "*Hello World!*" application.

- Start a new project in Microchip Studio
  File > New > Project… (CTRL+SHIFT+N)
- In the *New project* window:
  - Select GCC C Executable Project
  - Name: *HelloWorld*
  - Location: choose a convenient location on your hard drive
  - Click OK
- In the Device Selection window:
  - Select ATmega328p
  - Click OK
- The new project is created and the file main.c is opened.
- Replace the code in the file main.c by copy-and-pasting the following code.
  At this moment, it is not expected from you that you understand this code example. For now, just copy-and-paste this example as-is and the explanation follows later.

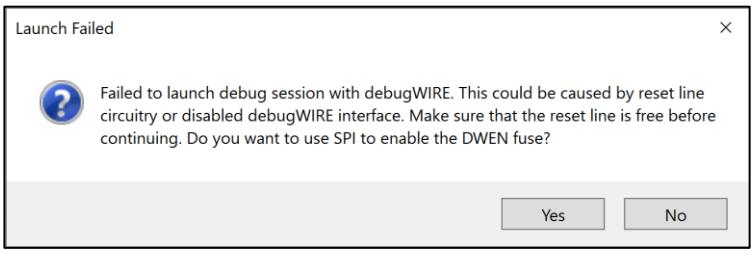
```c
/*
 * main.c
 *
 * Author : Hugo Arends
 */

#define F_CPU (16000000UL)

#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    // Configure PB5 so it will be an output pin
    DDRB |= (1<<DDB5);

    while (1)
    {
        // LED connected to PB5 on and wait
        PORTB |= (1<<PORTB5);
        _delay_ms(100);

        // LED connected to PB5 off and wait
        PORTB &= ~(1<<PORTB5);
        _delay_ms(900);
    }
}
```

- Select the debugger/programmer and the interface as follows: Project > HelloWorld Properties… (ALT+F7)



- Compile and upload the program to the microcontroller: Debug > Start Without Debugging (CTRL+ALT+F5).

If the following message pops up, click Yes:



FYI. By enabling the DWEN fuse, we enable the debugWire interface at the expense of the SPI interface. This means we cannot use the board in the Arduino IDE anymore. If you would like to revert to the SPI interface, start a debugging session (Debug > Start Debugging and Break (ALT+F7)) and disable the debugWire (Debug > Disable debugWire and Close). This restores the SPI interface.

- The executable is programmed to the microcontroller and after programming has finished, the LED connected to PB5 starts blinking as indicated in the source code.

5.3 Theory

The code example shown in the previous paragraph uses two registers, called DDRB and PORTB. Such registers are often referred to as DDRx and PORTx, because the microcontroller features several ports: port B, port C and port D. Each port controls up to eight digital pins. Therefore, each such register contains eight bits. Here is a visualization of the DDRB register, taken from the datasheet (Datasheet, 2015):

14.4.3 DDRB – The Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x04 (0x24)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
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	

The value of a bit is either logic 1 or logic 0. In the DDRB register, all eight bits have the logic value 0 after a microcontroller reset. If we would like to know what a logic 0 or logic 1 means for the hardware that is connected to a bit, we need to again consult the microcontroller's datasheet (Datasheet, 2015). The following description is given in paragraph 14.2.1 with respect to the DDRx register:

> *"The DDxn bit in the DDRx Register selects the direction of this pin. If DDxn is written logic one, Pxn is configured as an output pin. If DDxn is written logic zero, Pxn is configured as an input pin."*

Notice the use of *x* and *n* in this description:

- *x* should be replaced by a port (B, C or D)
- *n* should be replaced by a bit number (0 – 7)

From this information we can draw the conclusion that the DDRx register is used to set the direction of a digital pin to input (which is the default after reset) or to output. DDRx is the name of a register and DDxn is the name of a bit within that register.

The datasheet also provides information with respect to the PORTx register:

"If PORTxn is written logic one when the pin is configured as an output pin, the port pin is driven high (one). If PORTxn is written logic zero when the pin is configured as an output pin, the port pin is driven low (zero)."

And here is the visualisation of that register, taken from the datasheet (Datasheet, 2015):

14.4.2 PORTB – The Port B Data Register

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
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	

Now back to the example code. The LED on the Xplained mini board is connect to port B pin 5, or PB5 for short, as shown in the schematic. We should therefore use the DDRB register to configure PB5 as a digital output pin. This is achieved by writing logic 1 to bit number 5 of the DDRB register as we concluded from the information in the datasheet:

```
// Configure PB5 so it will be an output pin
DDRB |= (1<<DDB5);
```

Next is turning on the LED. This can be achieved by writing logic 1 to bit number 5 of the PORTB register.

```
// LED connected to PB5 on and wait
PORTB |= (1<<PORTB5);
```

And similarly, writing logic 0 to bit number 5 of the port B register will turn off the LED:

```
// LED connected to PB5 off and wait
PORTB &= ~(1<<PORTB5);
```

Don't worry if you do not fully understand the code examples presented here! These examples use several bitwise instructions that need a more thorough explanation. This topic will be covered next week. What is important, is that you get the basic idea of setting and resetting bits in registers.

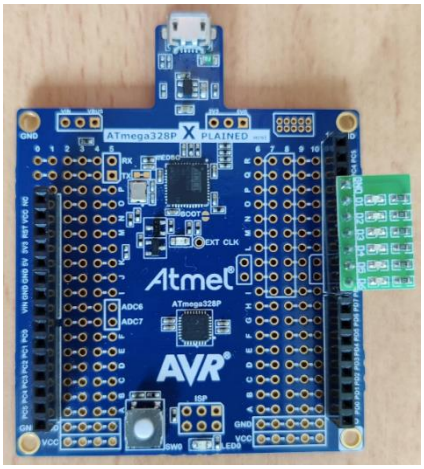
Finally, the datasheet provides several overviews of the ports and pins that are available in the microcontroller:

- Section 1 Pin Configurations shows the pinout for different packages.
- Paragraph 14.3.1, 14.3.2 and 14.3.3 with a detailed description of the options for a single pin.

How these pins map to the headers soldered on the Xplained mini board is shown in this study guide in section 3 Pin mapping reference.

5.4 Assignment

Use the code examples from the previous paragraph to fulfil the following assignment.



Create an LED chaser for six LEDs. Start with a new project and gradually add the functions as described below.

SMART technical specification		
#	**MoS CoW**	**Description**
T1	**M**	**The LED breakout board is connected to PB0-PB5 as shown in the adjacent image.**

SMART functional specification		
#	**MoS CoW**	**Description**
F1	**M**	**The LED connected to PB0 is the first LED that will turn on. The next LED is the LED connected to PB1. Then PB2, etc.**
F2	**M**	**Only a single LED is on at any moment in time.**
F3	**M**	**The interval between LEDs is 100ms. In other words, an LED is on for 100ms.**
F4	**M**	**After the last LED (PB5) turns off, the first LED (PB0) turns on immediately and the LED chaser starts over again from PB0.**

6 Week 2. Digital

6.1 Introduction

In order to fully understand the C code examples, it is important to understand bit manipulation instructions in the C programming language, a.k.a. I/O programming. The C programming course addresses this topic in general and this week we will emphasize on the concepts related to microcontrollers.

6.2 Theory

6.2.1 Bit manipulation

The register that will be used as an example is DDRB. Here is a visual representation of this register, taken from the datasheet (Datasheet, 2015):

14.4.3 DDRB – The Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x04 (0x24)	**DDB7**	**DDB6**	**DDB5**	**DDB4**	**DDB3**	**DDB2**	**DDB1**	**DDB0**	**DDRB**
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	

The visual representation shows:

- The name of the register: **The Port B Data Direction Register**
- The abbreviation of the register: **DDRB**
- The number for each bit position: **7** to **0**
- The abbreviation of each bit: **DDB7** to **DDB0**
- If a bit can be **read** and/or **written**
- The **initial logic value** after microcontroller reset

Suppose that at some moment in time the value of the DDRB register is equal to 0b00001111. And that in the application, code is executed that needs to write logic 0 to bit position 0. The following assignment in the C programming language can be used to achieve this:

```
DDRB = 0;
```

However, the number 0 in de decimal numbering system is equal to 0b00000000 in the binary numbering system. So the side effect of this assignment is that all other bits in register DDRB will be cleared as well!

The example shows the disadvantage of simply assigning the constant value (zero in this case) to a register, because all other bits are also set to logic 0. This is why bit manipulations are important. Data in a microcontroller is always read and written in parallel, even when there is a single bit of interest. Logic and bitwise operators are used for masking the bits that should not be updated when writing to a register, or when reading from a register when there is only a single bit of interest.

A mask is used to make sure that only the bits of interest are read or written. Here is an example of an 8-bit mask with two bits set to logic 1 and the other to logic 0:

```
MASK = 0b10000001;
```

The operator used with such a mask determines the result of the operation. Four operators are of interest and are discussed below: bitwise-or, bitwise-and combined with bitwise-not, and bitwise-exor.

6.2.1.1 Individual bit set: bitwise-OR

The bitwise-or operator can be used to set individual bits.

Sets the bits in DDRB that are set in MASK, all other bits are unchanged

```
DDRB |= MASK;
```

Example evaluation, when:

- DDRB is equal to 0b00001111
- The MASK is equal to 0b10000001

```
    DDRB |= MASK;
≡   DDRB = DDRB | MASK;
≡   DDRB = 0b00001111 | 0b10000001;
≡   DDRB = 0b10001111;
```

6.2.1.2 Individual bit clear: bitwise-AND with bitwise-not

The bitwise-and operator combined with the bitwise-not operator can be used to reset individual bits.

Clears the bits in DDRB that are set in MASK, all other bits are unchanged

```
DDRB &= ~MASK;
```

Example evaluation, when:

- DDRB is equal to 0b00001111
- The MASK is equal to 0b10000001

```
    DDRB &= ~MASK;
≡   DDRB =    DDRB    & ~MASK;
≡   DDRB = 0b00001111 & ~0b10000001;
≡   DDRB = 0b00001111 &  0b01111110;
≡   DDRB = 0b00001110;
```

6.2.1.3 Individual bit toggle: bitwise-XOR

The bitwise-xor operator can be used to toggle individual bits.

Toggles the bits in DDRB that are set in MASK, all other bits are unchanged

```
DDRB ^= MASK;
```

Example evaluation, when:

- DDRB is equal to 0b00001111
- The MASK is equal to 0b10000001

```
    DDRB ^= MASK;
≡   DDRB =    DDRB    ^ MASK;
≡   DDRB = 0b00001111 ^ 0b10000001;
≡   DDRB = 0b10001110;
```

6.2.2 Bit checking

For checking the status of a single bit, a mask is used to ignore all bits not of interest. Let's for example check if bit 7 in register PINB is set or not. For that specific bit, we must use a mask with the 7th bit set and all other bits cleared: 0b10000000.

```
if((PINB & 0b10000000) != 0)
{
    // Yes, bit 7 is set
}
else
{
    // No, bit 7 is not set
}
```

Example evaluation 1, when:

- PINB is equal to 0b10001111
- The MASK is equal to 0b10000000

```
    if((   PINB    & 0b10000000) != 0)
≡   if((0b10001111 & 0b10000000) != 0)
≡   if((      0b10000000       ) != 0)
≡   if(             True             )
```

Example evaluation 2, when:

- PINB is equal to 0b00001111
- The MASK is equal to 0b10000000

```
    if((   PINB    & 0b10000000) != 0)
≡   if((0b00001111 & 0b10000000) != 0)
≡   if((      0b00000000       ) != 0)
≡   if(             False            )
```

Notice from both example evaluations how the mask and the bitwise-and operator make sure that all bits that are not of interest will evaluate to logic 0 for sure!

It is easy to check if a bit is cleared:

```
if((PINB & 0b10000000) == 0)
{
    // Yes, bit 7 is cleared
}
else
{
    // No, bit 7 is not cleared
}
```

Often you will run into code examples that do not clearly show a condition in the if-statement. Here is an example:

```
if(PINB & 0b10000000)
```

Embedded software engineers writing such code take advantage of the fact that in the C programming language the number **0 evaluates to false** and that **any other number evaluates to true**. Try for yourself, for example, how this if-statement evaluates if PINB is equal to 0b10001111 and if PINB is equal to 0b00001111.

6.2.3 Creating masks

Masks are often created by using the bitwise left-shift operator, instead of hard coding. This makes the code more readable. Have a look at the following hard coded example. The result is that bit number 5 in the DDRB register will be written to logic 1:

```
DDRB |= 32; // 32 in decimal notation is equal to 0b00100000 in binary notation
```

The same result is achieved with the following instruction, which makes it easier to see that we are setting the 5th bit in register DDRB:

```
DDRB |= (1<<5); // (0b00000001 << 5) = 0b00100000
```

To make things even better, the microcontroller vendor offers a header file that maps a bit number to the abbreviation as mentioned in the datasheet. Remember that the datasheet shows the following description for the DDRB register in paragraph 14.4.3:

14.4.3 DDRB – The Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x04 (0x24)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
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	

This tells us that we can also use the following instruction, which makes the code even more readable:

```
DDRB |= (1<<DDB5);
```

Another advantage is that the code is more portable across devices. If a bit is located at another position in a register in a different microcontroller, it only requires updating the header file that maps the abbreviations to the actual bit numbers.

Finally, masks can have multiple bits set to logic 1. Here is an example that sets both the 5th and the 3rd bit by using the bitwise-or operator:

```
DDRB |= ((1<<DDB5) | (1<<DDB3));
```

This instruction evaluates as follows, when:

- DDRB is equal to 0b00001111

```
    DDRB |= ((1<<DDB5) | (1<<DDB3));
≡   DDRB =    DDRB    | (( 1<<DDB5  ) | ( 1<<DDB3   ));
≡   DDRB = 0b00001111 | ((   1<<5   ) | (   1<<3    ));
≡   DDRB = 0b00001111 | ((0b00100000) | (0b000001000));
≡   DDRB = 0b00001111 | (        0b00101000          );
≡   DDRB = 0b00101111;
```

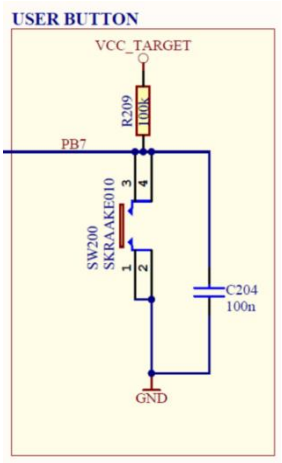
6.2.4 Input pins

As seen last week, the DDRx register is used to configure a pin as a digital input or as a digital output pin. This week we will again use PB5 as an output pin and additionally use PB7 as an input pin. The

Xplained mini board already features a switch, that is connected to pin PB7. The hardware connection is shown in more detail in the adjacent image, taken from the user guide (User Guide, 2017). Notice that:

- if the switch is not pressed, the value at PB7 will read logic 1;
- if the switch is pressed, the value at PB7 will read logic 0.

The following example code shows how to read the logic value from pin PB7. Notice that it introduces a new register called PINB. The PINx register can be used to read the logic value of a port pin.



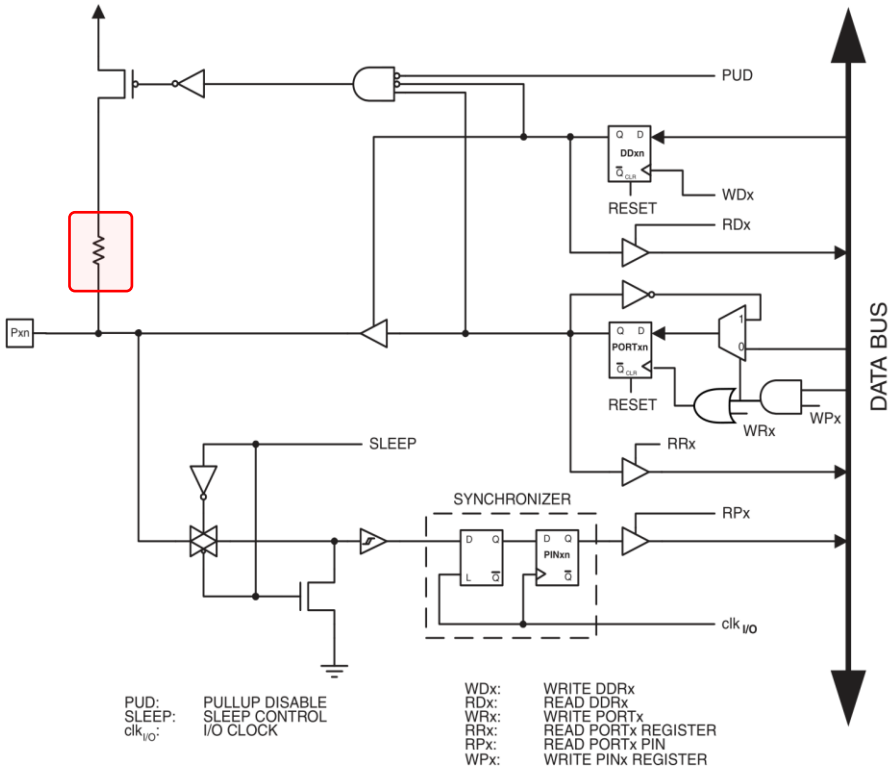

```
#define F_CPU (16000000UL)

#include <avr/io.h>

int main(void)
{
    // Configure PB5 so it will be an output pin
    DDRB |= (1<<DDB5);

    // Configure PB7 so it will be an input pin
    // Notice that strictly speaking this instruction is redundant,
    // because after a reset, the initial value of this bit is already
    // logic 0.
    DDRB &= ~(1<<DDB7);

    while (1)
    {
        // Read the content from the PINB register and perform a bitwise-and
        // operation using the mask (1<<PINB7). Only if bit number 7 in the
        // PINB register is not set, the condition evaluates to True.
        //
        // Notice that when the switch is not pressed PB7 reads logic 1!
        // In other words: this if-statement checks if the switch is pressed.
        if((PINB & (1<<PINB7)) == 0)
        {
            // LED connected to PB5 on
            PORTB |= (1<<PORTB5);
        }
        else
        {
            // LED connected to PB5 off
            PORTB &= ~(1<<PORTB5);
        }
    }
}
```

6.2.5 Pull-up resistors

The schematic as shown on this page shows a resistor R209 that is connected to VCC_TARGET. This resistor makes sure that a valid logic level is available on pin PB7 when the switch is not pressed. After all, PB7 would be a so-called ‘floating input’ if R209 didn’t exist (and the switch is not pressed).

Microcontroller vendors offer an option to enable pull-up resistors that are built-in inside the microcontroller. The ATmega328p also has this option for each I/O pin, as shown in the block diagram, taken from the datasheet (Datasheet, 2015).

Figure 14-2. General Digital I/O(1)



The datasheets also describes how to enable such a pullup resistor:

> *"If PORTxn is written logic one when the pin is configured as an input pin, the pull-up resistor is activated."*

The following code example shows how the pull-up resistors are enabled for pins PD3 to PD6.

```
#define F_CPU (16000000UL)

#include <avr/io.h>

int main(void)
{
	// PB0 to PB5 logic 1: configures these pins as output
	DDRB |= (1<<DDB5) | (1<<DDB4) | (1<<DDB3) |
		(1<<DDB2) | (1<<DDB1) | (1<<DDB0);

	// PD7 logic 1: configures this pin as output and leave all other bits
	// unchanged (logic 0)
	DDRD |= (1<<DDD7);

	// PD7 logic 0: pin value is equal to GND
	PORTD &= ~(1<<PORTD7);

	// PD3 to PD6 logic 1: enable pull-up resistors, because by default, these
	// pins are configured as input pins
	PORTD |= (1<<PORTD6) | (1<<PORTD5) | (1<<PORTD4) | (1<<PORTD3);
```

```
        while (1)
        {
                // Read the pin PD3 to PD6 values and mask all other PORTD pins
                unsigned char switches = PIND & 0b01111000;

                // Check if the switch connected to PD3 was pressed or not
                if(switches & (1<<PIND3))
                {
                        // Switch not pressed, so set PB0 to logic 0 (GND).
                        // This turns off the LED connected to PB0.
                        PORTB &= ~(1<<PORTB0);
                }
                else
                {
                        // Switch pressed, so set PB0 to logic 1 (Vcc = 5V).
                        // This turns on the LED connected to PB0.
                        PORTB |= (1<<PORTB0);
                }
        }
}
```

6.2.6 Summary

Three registers are associated with each digital I/O pin:

- **DDRx**: The Port x Data Direction Register
- **PORTx**: The Port x Data Register
- **PINx**: The Port x Input Pins

Table 14. in the datasheet (Datasheet, 2015) provides an overview of the configuration options. Notice the additional bit called PUD (located in the MCUCR register) that can be used to globally enable/disable all pull-up resistors.

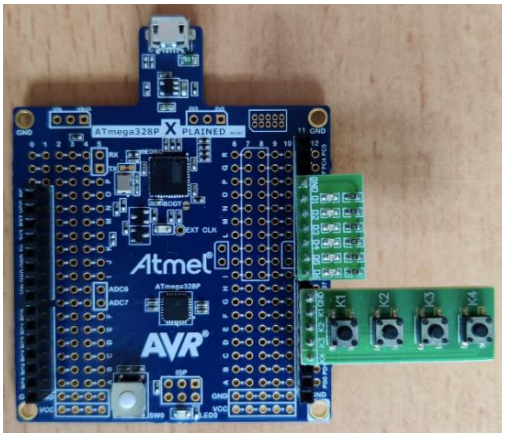
DDxn	PORTxn	PUD (in MCUCR)	I/O	Pull-up	Comment
0	0	X	Input	No	Tri-state (Hi-Z)
0	1	0	Input	Yes	Pxn will source current if ext. pulled low.
0	1	1	Input	No	Tri-state (Hi-Z)
1	0	X	Output	No	Output low (sink)
1	1	X	Output	No	Output high (source)

6.3 Assignment

Use the code examples from the previous paragraph to fulfil the following assignment.

Create an LED chaser for six LEDs that has several functions that can be selected by pressing a switch. Start with a new project and gradually add the functions as described below.

SMART technical specification		
#	**MoS CoW**	**Description**
T1	**M**	**The LED breakout board is connected to PB0-PB5.** **The LED connected to PB0 is considered the first LED (see adjacent image).**
T2	**M**	**The switches breakout board is connected to PD3-PD7 (see adjacent image).** *Make sure that PD7 is configured as a digital output pin and that this pin is set to logic 0 (see code example in paragraph 6.2.5 Pull-up resistors). Make sure to never set this pin to logic 1 in the rest of the code.*



SMART functional specification		
#	**MoS CoW**	**Description**
F1	**M**	**When the application starts:**
F1.1	M	The LED chaser moves from PB0 to PB5.
F1.2	M	As soon as PB5 switches off, PB0 will switch on.
F1.3	M	The initial interval is 100ms.
F2	**M**	**The switches K1 to K4 control the speed and direction of the LED chaser.**
F2.1	M	K1: interval increases by 100ms. The maximum interval is 1000ms.
F2.2	M	K2: interval decreases by 100ms. The minimum interval is 100ms.
F2.3	M	K3: the LED chaser moves from PB0 to PB5.
F2.4	M	K4: the LED chaser moves from PB5 to PB0.

TIP. The function `_delay_ms()` requires a compile time integer constant. This means that the following code example will generate a compile error:

```
/*
 * main.c
 *
 * Created: 4-6-2021 12:22:38
 * Author : Hugo Arends
 */

#define F_CPU (16000000UL)

#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
```

```
    // PB5 output
    DDRB |= (1<<DDB5);

    // PB7 input
    DDRB &= ~(1<<DDB7);

    // Initial delay value
    int delay_in_ms = 100;

    while (1)
    {
        // Toggle the LED connected to PB5
        PORTB ^= (1<<PORTB5);

        // Check if the pin is pressed
        if((PINB & (1<<PINB7)) == 0)
        {
            // Increment the delay
            delay_in_ms += 100;

            if(delay_in_ms > 1000)
            {
                delay_in_ms = 1000;
            }

            // Debounce the switch
            _delay_ms(30);
        }

        // Wait
        _delay_ms(delay_in_ms);
    }
}
```

The highlighted line in red generates the following compiler error:

__builtin_avr_delay_cycles expects a compile time integer constant

This is caused by the fact that *delay_in_ms* is a variable and not a compile time constant. As indicated by the compiler error message, this is not allowed for the *_delay_ms()* function. Although less accurate, creating a variable delay can then be achieved by using a for-loop as follows:

```
        // Wait
        for(int i=0; i<delay_in_ms; i++)
        {
            _delay_ms(1);
        }
```

7 Week 3. Analog

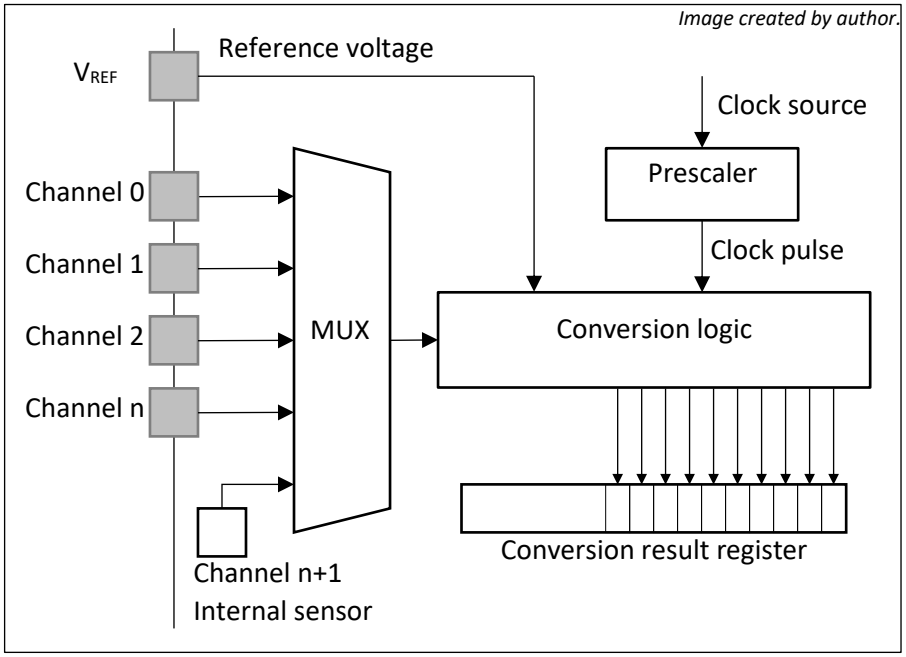
7.1 Introduction

So far we have seen logic signals, which are true (1, on, set, V_{cc}) or false (0, off, clear, GND). If we need to measure an analog signal, we can use an analog-to-digital convertor (ADC). An example of using the ADC is when a sensor measures some kind of quantity, such as distance or air pressure, and produces an output voltage that relates to the measured value.

7.2 Theory

The basic operation of an ADC is depicted in the adjacent block diagram. Most ADCs have multiple input channels. These input channels are connected to different I/O pins or connected to an internal sensor, such as a temperature sensor. A multiplexer (MUX) is used to select one channel at a time. Bits in a configuration register (not shown in in the block diagram) control this MUX and hence determine which channel is selected.

Image created by author.



The conversion logic converts the analog value on one of the channels to a digital value. The number of bits of the digital value are called the ADC's resolution (n). The block diagram shows, as an example, a 10-bit resolution and that the result must be read from a 16-bit conversion result register.

An analog voltage on an input channel is measured with respect to a reference voltage. The conversion result is calculated with the equation

$$ADC\ digital\ conversion\ result = \frac{V_{channel}}{V_{REF}} \times (2^n - 1)$$

where $V_{channel}$ is the voltage on the selected input channel, V_{REF} is the reference voltage and n is the ADC's resolution.

Each conversion requires the conversion logic to execute a number of steps. These steps are executed at the pace of the clock pulse. A prescaler is used to adjust the speed of a conversion. A flag in a status register (not shown in the block diagram) can be checked to see if a conversion is ready. As soon as a conversion is ready, another channel can be selected and a new conversion can be started.

Most microcontroller vendors offer additional ADC features, such as configurable reference voltages, configurable resolution, or even more than one on-chip ADCs.

7.3 Code examples

7.3.1 Polling

The following code example demonstrates how to initialize the ADC, start a conversion, wait for the conversion to complete and use the LED breakout board to visualize the conversion result.

```
#define F_CPU (16000000UL)

#include <avr/io.h>

int main(void)
{
    // PB0 - PB5 output, rest input
    DDRB = 0b00111111;

    // Initialize the ADC peripheral
    // ADC Multiplexer Selection Register:
    //   REFS[1:0] = 01   : AV_CC with external capacitor at AREF pin
    //   ADLAR     = 0    : Right adjust result
    //   MUX[3:0]  = 0000 : Select channel ADC0
    ADMUX = (1<<REFS0);

    // ADC Control and Status Register A:
    //   ADEN      = 1    : ADC enable
    //   ADSC      = 0    : Do not start a conversion
    //   ADATE     = 0    : Auto triggering disabled
    //   ADIF      = 0    : Do not clear the interrupt flag
    //   ADIE      = 0    : Do not enable interrupts
    //   ADPS[2:0] = 111 : Prescaler division factor 128
    ADCSRA = (1<<ADEN) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0);

    while (1)
    {
        // Start a conversion (making sure all other bits do not change)
        ADCSRA |= (1<<ADSC);

        // Wait for the conversion to complete by polling the ADSC bit
        while((ADCSRA & (1<<ADSC)) != 0)
        {;}

        // Read the result
        uint16_t val = ADC;

        // Visualize the result
        if      (val < 146) { PORTB = 0b00000000; }
        else if (val < 293) { PORTB = 0b00000001; }
        else if (val < 439) { PORTB = 0b00000011; }
        else if (val < 585) { PORTB = 0b00000111; }
        else if (val < 731) { PORTB = 0b00001111; }
        else if (val < 878) { PORTB = 0b00011111; }
        else                { PORTB = 0b00111111; }

    }
}
```

The example shows that three registers are required for this polling based AD conversion. Polling means that the program actively waits for a conversion to complete. This example uses a while-loop to do the waiting. The while-loop constantly reads the ADCSRA register and uses the mask (1<<ADSC) to check the value of only the ADSC bit. As long as this ADSC bit is set, the application needs to wait,

because the conversion is still ongoing. As soon as the bit is not set anymore, the condition evaluates to false and the application continues.

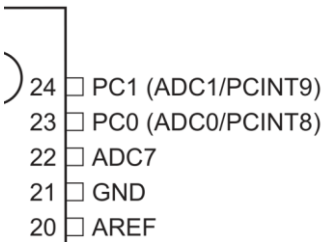
The description of all the bits in the ADMUX and ADCSRA register is provided in the datasheet (Datasheet, 2015) and will not be repeated here. In order to setup these registers, you should be aware of the following:

- The ATmega328p Xplained mini board operates with V_{cc} at 5V. This means that V_{ref} is also equal to 5V, at least when the REFS[1:0] bits are set to 0b01.
- The ADC requires a specific clock frequency. The following is mentioned in the datasheet in paragraph 24.4:

  > "*By default, the successive approximation circuitry requires an input clock frequency between 50kHz and 200kHz to get maximum resolution. If a lower resolution than 10 bits is needed, the input clock frequency to the ADC can be higher than 200kHz to get a higher sample rate.*"

  This means that we must tune the prescaler bits ADPS[2:0] to the frequency of the ADC clock source. The default ADC clock source frequency is equal to the core clock (F_CPU) which is 16 MHz. The prescaler **must** therefore be a value between 80 and 320.

- The pin configurations in the datasheet (Datasheet, 2015) show which port pin is connected to which ADC channel. The following (partial) package pinout, for example, shows that if you would like to use ADC channel 0 (ADC0), it means that you must use port pin PC0.



7.3.2 Interrupt

Instead of waiting for a conversion to complete with a while-loop, we can setup the ADC to constantly produce ADC conversions and notify the application of a conversion complete by triggering an Interrupt Service Routine (ISR). Interrupts are a fundamental mechanism of microcontrollers and allows for event driven applications. You can think of an ISR as a function that is executed as soon as a specific hardware event occurs. In an ADC, this hardware event is a completed conversion. An important side effect of interrupts is that the execution of the main application is postponed as long as an interrupt is being serviced.

Here is an example of and interrupt based implementation. Notice the global variable called 'val' that is written in the ISR() and read in main(). There is also an additional register, ADCSRB, that selects the trigger source for a new conversion. Refer to the datasheet to find out what 'Free Running mode' exactly means and what other triggers sources are available.

```
#define F_CPU (16000000UL)

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

// Global variable for the ADC conversion result
volatile uint16_t val=0;
```

```
// This is the Interrupt Service Routine (ISR). This 'function' is executed
// every time an ADC conversion is complete, because we configured it
// that way in main().
// You cannot choose the name of an ISR. These are predefined in the header
// file called iom328p.h.
ISR(ADC_vect)
{
    // Read and update the result
    val = ADC;
}

int main(void)
{
    // PB0 - PB5 output, rest input
    DDRB = 0b00111111;

    // Initialize the ADC peripheral
    // ADC Multiplexer Selection Register:
    //   REFS[1:0] = 01   : AV_CC with external capacitor at AREF pin
    //   ADLAR     = 0    : Right adjust result
    //   MUX[3:0]  = 0000 : Select channel ADC0
    ADMUX = (1<<REFS0);

    // ADC Control and Status Register A
    //   ADEN      = 1    : ADC enable
    //   ADSC      = 1    : Start the first conversion
    //   ADATE     = 1    : Auto triggering enabled
    //   ADIF      = 1    : Clear the interrupt flag (in case there was any)
    //   ADIE      = 1    : Enable interrupts
    //   ADPS[2:0] = 111 : Prescaler division factor 128
    ADCSRA = 0b11111111;

    // ADC Control and Status Register B:
    //   ADTS[2:0] = 000 : Free Running mode
    ADCSRB = 0;

    // Enable interrupts
    sei();

    while (1)
    {
        // Visualize the latest conversion result
        if      (val < 146) { PORTB = 0b00000000; }
        else if (val < 293) { PORTB = 0b00000001; }
        else if (val < 439) { PORTB = 0b00000011; }
        else if (val < 585) { PORTB = 0b00000111; }
        else if (val < 731) { PORTB = 0b00001111; }
        else if (val < 878) { PORTB = 0b00011111; }
        else                { PORTB = 0b00111111; }
    }
}
```

7.4 Assignment

Use the code examples from the previous paragraph to fulfil the following assignment.

Read the value from two (fake) sensors. The sensors are imitated by two potentiometers. If both sensors are above a certain value, an alarm should be raised visualized by an LED. The alarm is turned off by clicking a switch, which will turn off the LED.

SMART technical specification		
#	**MoS CoW**	**Description**
T1	**M**	**Two sensors (potentiometers in the range of 1 kΩ to 100 kΩ) are connected to channel ADC0 and ADC1.**
T2	**M**	**Use the switch that is already available on the ATmega328p Xplained mini board.**

SMART functional specification		
#	**MoS CoW**	**Description**
F1	**M**	**When the application starts, The LED connected to PB5 is off.**
F2	**M**	**The two sensors activate the alarm.**
F2.1	M	Both sensor readings are mapped to a range of 0 to 100.
F2.2	M	Only if both sensor values are above 75, an alarm is activated.
F2.2.1	M	The LED connected to PB5 is turned on.
F2.2.2	M	The alarm does not deactivate if one or both sensors drop below 75.
F3	**M**	**The switch deactivates the alarm.**
F3.1	M	If the switch is pressed and an alarm was activated, the alarm will be deactivated. The LED connected to PB5 turns off.
F3.2	M	If the switch is pressed and no alarm was activated, nothing happens.

Tip. This assignment requires that you change to another channel after a conversion is complete. If you use the polling method, simply edit the ADMUX register before starting the next conversion. If you use the interrupt method, edit the ADMUX register in the ISR.

8 Week 4. Timers/Counters

8.1 Introduction

Most embedded software applications need some way for measuring time. A clock is an obvious example. Other examples are reading from sensors at a fixed interval or generating a waveform on a digital I/O pin. Microcontrollers have peripherals called timers/counters for this purpose.

8.2 Theory

Although timers vary greatly in features, the basic operation is the same for every timer. This basic operation is depicted in the following block diagram. To keep a clear overview, this figure does not show the control and status registers.

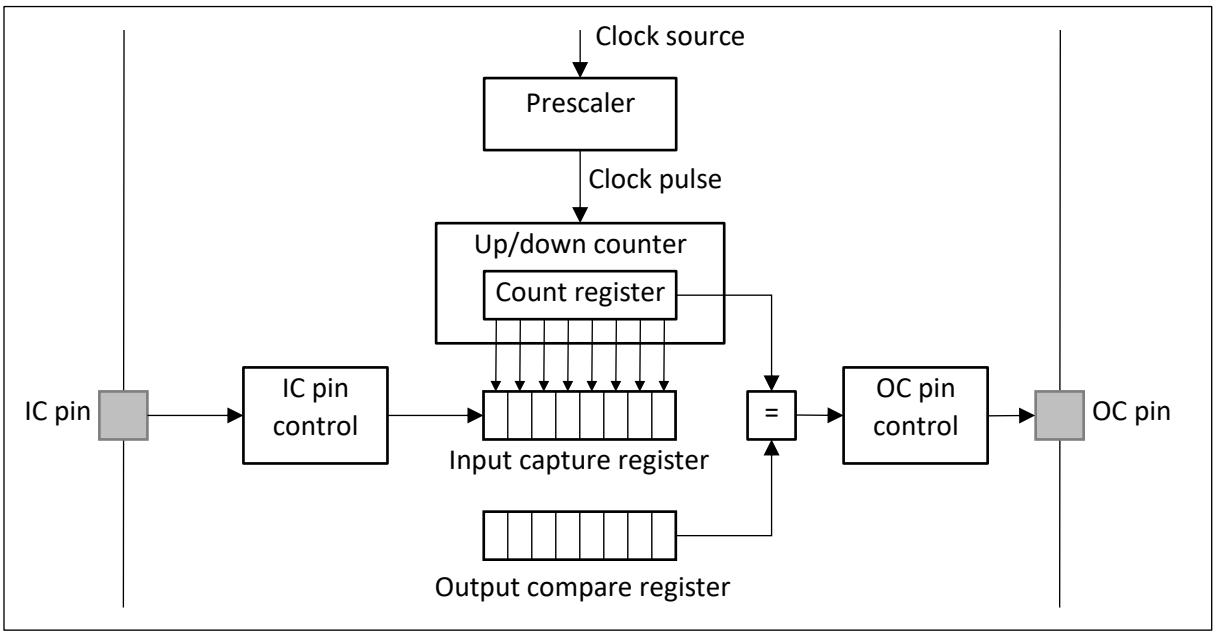


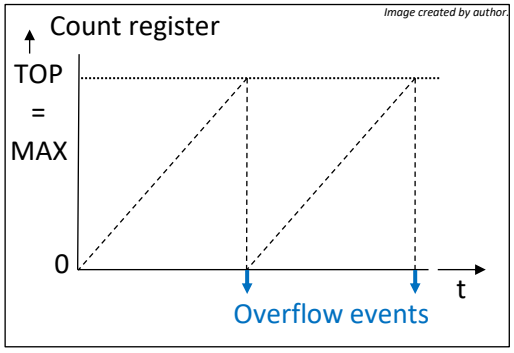
Image created by author.

The central part of the timer is the up/down counter. This hardware counts with single steps between zero and the timer's resolution. For an eight bit timer, the resolution is 256 steps at ranges from 0 to 255 (2^8-1). The speed of counting is determined by the clock pulse provided by a clock source. A prescaler is used to slow down the clock source, which allows to increase the time between timer events. The input capture (IC) pin can be used to capture the value of the count register at an external event. The output compare (OC) pin can be used to generate a waveform by comparing the output compare register to the count register.

Timers can basically be used for three modes of operations: normal, output compare and input capture.

8.2.1 Normal operation

This mode of operation is also known as *free running operation* or *timebase operation*. The timer simply counts from 0 to the counter's maximum value, and then restarts from 0 again. At the moment the count register switches from the TOP value to 0, a so-called overflow event is generated. Such an event can be used to trigger an ISR. The time between overflow events depends on the frequency of the clock source, the value of the prescaler (N), and the timer's maximum counting value (a.k.a. the timer's resolution).

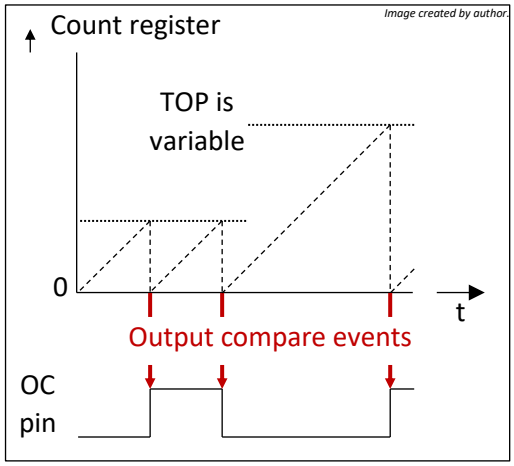


In some microcontrollers, a timer might count down in normal operation. Instead of an overflow event, it is called an underflow event.

8.2.2 Output compare operation

This operation is very similar to normal operation, except that the counter now restarts at 0 when it reaches the variable TOP value, which is stored in the output compare register. This operation, therefore, allows a more precise timing between events. In most microcontrollers, such an event can be used to trigger an ISR, but also to directly control a digital output pin. This can be used for generating digital signals, such as a square wave and PWM[1], by hardware instead of by software.

The time between output compare events depends on the frequency of the clock source, the value of the prescaler (N), and the TOP value stored in the output compare register.



For the ATmega328p microcontroller, the time between two events in normal mode and output compare mode is calculated with the following formula:

$$T_{events} = T_{F\_CPU} \times N \times (TOP + 1)$$

If TOP must be calculated for a specified time between events (T_{events}) and any given prescaler (N), this formula can be rewritten as follows:

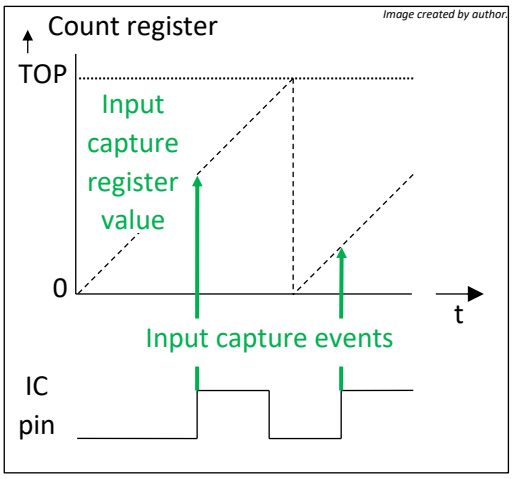
$$TOP = \frac{T_{events}}{T_{F\_CPU} \times N} - 1 \quad \text{and} \quad TOP = \frac{f_{F\_CPU}}{f_{events} \times N} - 1$$

Always make sure that the calculated TOP value fits within the 8-bit or 16-bit register!

[1] https://en.wikipedia.org/wiki/Pulse-width_modulation

8.2.3 Input capture operation

This operation captures the value of the count register when a rising edge, falling edge, or both is detected on the IC pin. This can be used by the embedded software engineer for measuring frequency, duty cycle, or other features of external signals. Most microcontroller vendors offer the possibility to execute an ISR on such an event. Notice that it is the embedded programmer's responsibility to make sure that the time between events is long enough for the processor to handle the captured values and that the counter's speed is not too fast to make sure that an event is captured within the timer's resolution.



8.3 Code examples

One thing to remember is that the ATmega328p microcontroller features three timers/counters:

- Timer/counter 0 is an 8-bit timer, which has a maximum count of 256 steps (from 0x00 to 0xFF).
- Timer/counter 1 is a 16-bit timer, which has a maximum count of 65536 steps (from 0x0000 to 0xFFFF).
- Timer/counter 2 is an 8-bit timer, which has a maximum count of 256 steps (from 0x00 to 0xFF).

8.3.1 Normal operation

Setting up a timer/counter in normal operation and generating interrupts at fixed intervals, requires the following steps:

- Set the *waveform generation mode* bits (WGM1[3:0]) to mode 0, as shown in the following table, taken from the datasheet (Datasheet, 2015):

Table 15-5. Waveform Generation Mode Bit Description(1)

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, phase correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, phase correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, phase correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, phase and frequency correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, phase and frequency correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, phase correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, phase correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

  These bits are located in two registers: TCCRnA and TCCRnB

- Setting the *clock select* bits (CSn[2:0]), located in register TCCRnB, to select one of eight prescalers.

- Enable overflow interrupts.
- Implement an overflow interrupt handler.

Here is an example:

```
#define F_CPU (16000000UL)

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

// Interrupt Service Routine that is automatically executed as soon as
// timer/counter 1 has reached its TOP value.
ISR(TIMER1_OVF_vect)
{
    // Toggle pin PB0
    // This is a special feature of a digital I/O pin.
    // For more information, refer to section 13.2.2 of the datasheet.
    PINB = (1<<PINB0);
}

int main(void)
{
    // PB0 - PB5 output, rest input
    DDRB = 0b00111111;

    // Configure Timer/counter 1 to generate an interrupt approximately every second
    //
    // For this configuration, several bits can keep their reset value.
    // This means that this function only implements instructions for bits
    // that must be updated.
    // Refer to the datasheet for a description of all the bits.

    // - WGM1[3:0] = 0000 : waveform generation mode is normal mode (default)
    // - CS1[2:0] = 100 : 256 prescaler
    // - TOP is fixed in normal mode = 65535
    //
    // T_events = T_CPU * N * (TOP + 1)
    //          = 1/16MHz * 256 * (65535 + 1)
    //          = 1.048576s
    TCCR1B |= (1<<CS12);

    // Timer/Counter1, Overflow Interrupt Enable
    TIMSK1 |= (1<<TOIE1);

    // Global interrupt enable
    sei();

    while (1)
    {
        ; // Do nothing
    }
}
```

8.3.2 Output compare operation

Output compare mode is called *Clear Timer on Compare Match (CTC) Mode* in the ATmega328p microcontroller. The WGM1[3:0] bits can be used to select two waveform generations modes that support this operation. These are modes 4 and 12. The difference between the two modes is the register that holds the TOP value.

The first example shows how to setup timer 0 in CTC mode to generate an interrupt every 10 ms. With this requirement, together with a given F_CPU of 16 MHz, we can calculate the desired TOP. However, we should also decide on a prescaler (N), otherwise the equation for calculating the TOP value has two unknowns:

$$TOP = \frac{T_{events}}{T_{F_{CPU}} \times N} - 1 =$$

Let's pick N=1:

$$TOP = \frac{0.01\ s}{\frac{1}{16\ MHz} \times 1} - 1 = 159999$$

This value should then be written in the register OCR0A (assuming mode 4 is selected). The datasheet, however, shows that this is an 8-bit register:

14.9.4 OCR0A – Output Compare Register A

Bit	7	6	5	4	3	2	1	0	
0x27 (0x47)	OCR0A[7:0]								OCR0A
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	

The output compare register A contains an 8-bit value that is continuously compared with the counter value (TCNT0). A match can be used to generate an output compare interrupt, or to generate a waveform output on the OC0A pin.

This means that the value 159999 does not fit and hence we cannot use a prescaler set to 1!

Now instead, let's pick the largest possible prescaler N=1024:

$$TOP = \frac{0.01s}{\frac{1}{16MHz} \times 1024} - 1 = 156.25$$

This value does fit in an 8-bit register.

```
OCR0A = 156;
```

Notice that this register takes an integer value. This means that due to rounding, we are creating a (slightly) different timing than originally intended.

```
// Interrupt Service Routine that is automatically executed as soon as
// timer/counter 0 has reached its TOP value.
ISR(TIMER0_COMPA_vect)
{
    // For visualization purpose, let's toggle the pin every 10th time
    // this ISR is called.
    // The keyword static ensures that the value of the variable cnt
    // will not have changed when the ISR is executed the next time.
    static unsigned char cnt = 0;

    cnt++;

    if(cnt > 9)
    {
        cnt = 0;

        // Toggle pin PB0
        // This is a special feature of a digital I/O pin.
        // For more information, refer to section 13.2.2 of the datasheet.
        PINB = (1<<PINB0);
    }
```

```
}

int main(void)
{
    // PB0 - PB5 output, rest input
    DDRB = 0b00111111;

    // Configure PB7 so it will be an input pin
    // Notice that strictly speaking this instruction is redundant,
    // because after a reset, the initial value of this bit is already
    // logic 0.
    DDRB &= ~(1<<DDB7);

    // Configure Timer/counter 0 to generate an interrupt approximately every 10ms.
    //
    // For this configuration, several bits can keep their reset value.
    // This means that this function only implements instructions for bits
    // that must be updated.
    // Refer to the datasheet for a description of all the bits.

    // - WGM0[2:0] = 010 : waveform generation mode is CTC with TOP in OCR0A
    // - CS0[2:0] = 101 : 1024 prescaler
    // - TOP = 156
    //
    // T_events = T_CPU * N * (TOP + 1)
    //          = 1/16MHz * 1024 * (156 + 1)
    //          = 0.010048s
    TCCR0A |= (1<<WGM01);
    TCCR0B |= (1<<CS02) | (1<<CS00);
    OCR0A = 156;

    // Timer/Counter0 Output Compare Match A Interrupt Enable
    TIMSK0 |= (1<<OCIE0A);

    // Global interrupt enable
    sei();

    while (1)
    {
        ; // Do nothing
    }
}
```

8.3.3 Using output compare to create a millis() function

Output compare mode can be used to keep track of time. This paragraph shows an example of how to setup timer/counter 0 and implements an ISR to increment a global variable every millisecond. The global variable can then be used in the main application to find out how much milliseconds have passed since the microcontroller started.

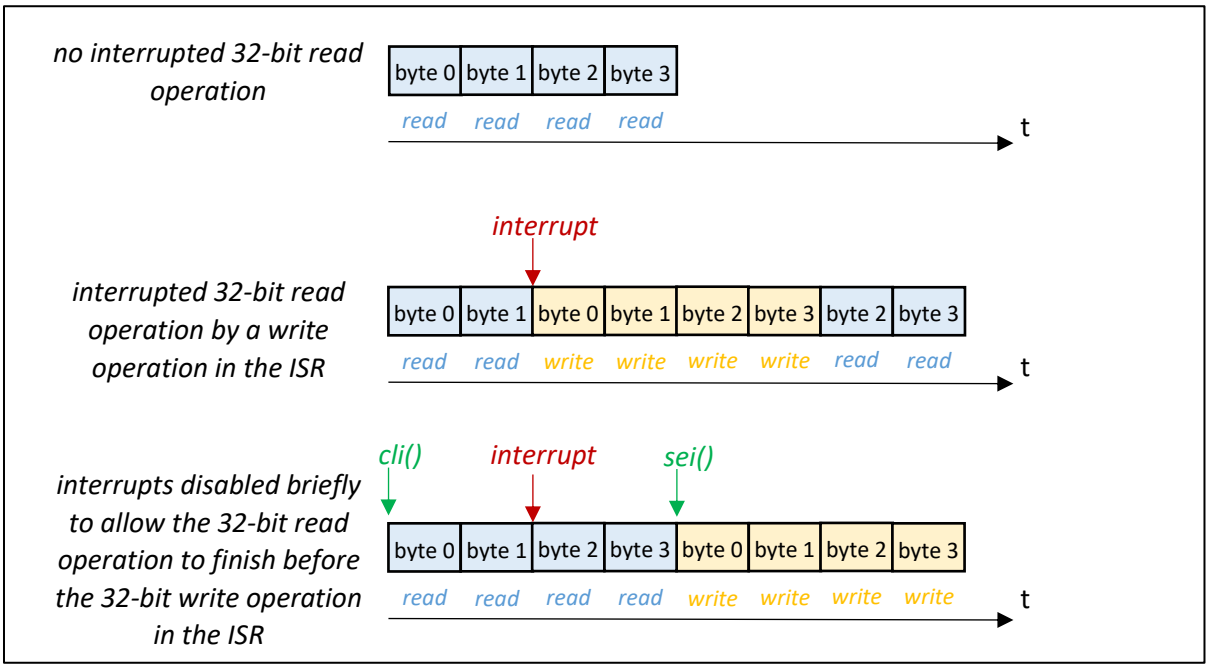
At a F_CPU frequency of 16MHz and a prescaler of 64, the following TOP value is applicable:

$$TOP = \frac{0.001s}{\frac{1}{16MHz} \times 64} - 1 = 249$$

A prescaler of 64 is selected, because then the TOP value isn't a fraction and fits within an eight bit range.

Every time the ISR is executed, a global variable called *ms* will be incremented by one. This global variable is stored in a 32-bit variable (uint32_t). This allows the application to count up-and-until $2^{32} - 1$ milliseconds, which is equal to 49.71 days.

There is a catch when reading variables that are stored in a multiple of eight bits. The reason for that, is that in an 8-bit microcontroller it takes multiple actions to read or write a 32-bit value. This is depicted in the following timing diagrams.



The function `uint32_t millis(void)` disables interrupts briefly when reading the global variable *ms*. This is also known as a critical section to allow for atomic data access.

The following code example shows how to initialize timer/counter 0 and create a function to read the *ms* global variable making sure that a read operation cannot be interrupted.

```
#define F_CPU (16000000UL)

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

volatile uint32_t ms = 0;

void millis_init(void)
{
    ms = 0;

    // Configure Timer/counter 0 to generate an interrupt every millisecond
    //
    // - WGM0[2:0] = 010 : waveform generation mode is CTC with TOP in OCR0A
    // - CS0[2:0] = 011 : 64 prescaler
    // - TOP : 249
    //
    // T_events = T_CPU * N * (TOP + 1)
    //          = 1/16 MHz * 64 * (249 + 1)
    //          = 1 ms
    TCCR0A |= (1<<WGM01);
    TCCR0B |= (1<<CS01) | (1<<CS00);
    OCR0A = 249;
```

```
    // Timer/Counter0 Output Compare Match A Interrupt Enable
    TIMSK0 |= (1<<OCIE0A);
}

// Interrupt Service Routine that is automatically executed as soon as
// timer/counter 0 has reached its compare value
ISR(TIMER0_COMPA_vect)
{
    ms++;
}

inline uint32_t millis(void)
{
    // ms is a 32-bit variable (uint32_t). This means that multiple accesses
    // are needed to read or write the value of ms. There is a chance that
    // in the middle of these multiple accesses, the ms value is written due to
    // the ISR being triggered. In order to make sure the value of ms is not
    // updated when reading it, disable interrupts while reading the value.
    cli();
    uint32_t ret = ms;
    sei();

    return ret;
}

int main(void)
{
    // PB5 output
    DDRB |= (1<<PORTB5);

    // PB7 input
    DDRB &= ~(1<<DDB7);

    // Initialize the millisecond counter
    millis_init();

    // Global interrupt enable
    sei();

    uint32_t previousmillis = 0;
    uint32_t currentmillis = 0;
    uint32_t interval = 100;

    while(1)
    {
        // Get the current millis
        currentmillis = millis();

        // Has 'interval' time passed since the last time we checked?
        if((currentmillis - previousmillis) >= interval)
        {
            // Yes, 'interval' time has passed
            // Save the current millis
            previousmillis = currentmillis;

            // Toggle the LED
            PORTB ^= (1<<PORTB5);
        }
    }
}
```

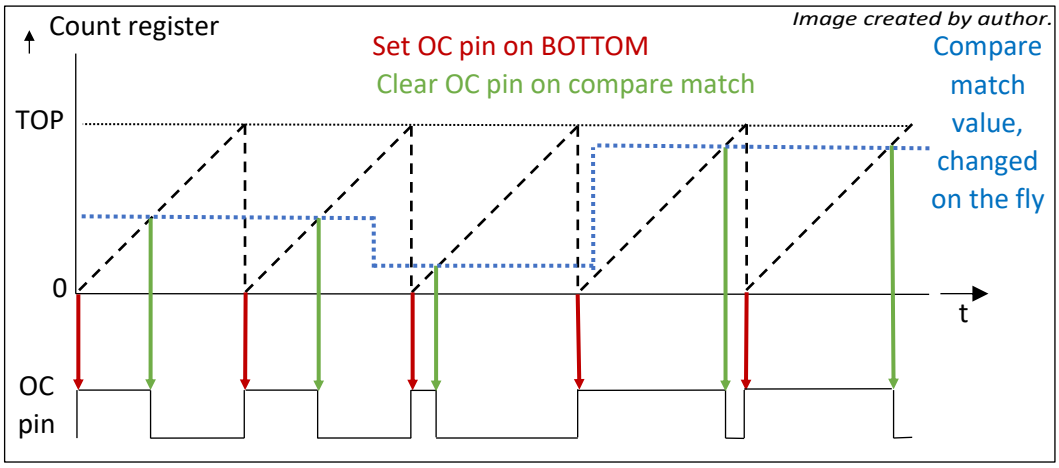
8.3.4 PWM

Output compare modes can also be used for generating PWM signals. Instead of toggling a pin in software, the timer/counter has built-in features for toggling pins. This doesn't require the use of any interrupt at all. The following table shows the bits of interest for generating a so-called fast-PWM signal:

Table 15-3. Compare Output Mode, Fast PWM(1)

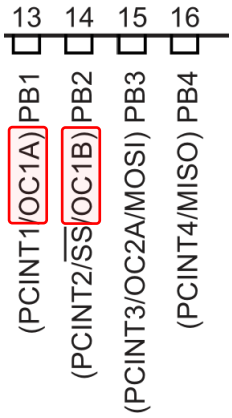
COM1A1/COM1B1	COM1A0/COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	WGM13:0 = 14 or 15: Toggle OC1A on compare match, OC1B disconnected (normal port operation). For all other WGM1 settings, normal port operation, OC1A/OC1B disconnected.
1	0	Clear OC1A/OC1B on compare match, set OC1A/OC1B at BOTTOM (non-inverting mode)
1	1	Set OC1A/OC1B on compare match, clear OC1A/OC1B at BOTTOM (inverting mode)

The following timing diagram shows the interaction of the TOP value, the compare match value and the resulting signal on the OC pin when the bits COM1x[1:0] are set to 0b10.



From this timing diagram we conclude that the PWM frequency is determined by the TOP value. The value in the compare register (= compare value) sets the duty cycle, which should be a value between 0 and TOP.

The following code example shows how to create a 50 Hz PWM signal. The register with the TOP value is ICR1, see the setting of the WGM1[3:0] bits. This means that we can generate a PWM signal on both channels A and B simultaneously. The OC pins are dedicated to a channel. The pin configurations in the datasheet (Datasheet, 2015) show which port pins are connected to OC1A and OC1B:




```c
#define F_CPU (16000000UL)

#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    // PB0 - PB5 output, rest input
    DDRB = 0b00111111;

    // Configure Timer/counter 1 to generate an interrupt approximately every second
    //
    // For this configuration, several bits can keep their reset value.
    // This means that this function only implements instructions for bits
    // that must be updated.
    // Refer to the datasheet for a description of all the bits.

    // - WGM1[3:0] = 1110 : waveform generation mode is Fast PWM with TOP in ICR1
    // - CS1[2:0] = 011 : 64 prescaler
    //
    // TOP = (F_CPU / (F_PWM * N)) - 1
    //     = (16MHz / (50Hz * 64)) - 1
    //     = 4999
    TCCR1A |= (1<<WGM11);
    TCCR1B |= (1<<WGM13) | (1<<WGM12) | (1<<CS11) | (1<<CS10);
    ICR1 = 4999;

    // Set Compare Output Mode for both channels A and B
    // The mode is fast-PWM
    // COM1A[1:0] = 10 : Clear OC1A/OC1B on compare match, set OC1A/OC1B at
    //                   BOTTOM (non-inverting mode)
    // COM1B[1:0] = 11 : Set OC1A/OC1B on compare match, clear OC1A/OC1B at
    //                   BOTTOM (inverting mode)
    TCCR1A |= (1<<COM1A1) | (1<<COM1B1) | (1<<COM1B0);

    // Set the initial duty cycle for both channels to 50%
    OCR1A = 2500;
    OCR1B = 2500;

    uint16_t cnt = 0;

    while (1)
    {
        // Gradually update the duty cycle by changing the
        // compare values every millisecond.
        // This takes a total of 5000 * 1ms = 5000ms = 5s.
        _delay_ms(1);
```

```
        cnt++;

        if(cnt > 4999)
        {
            cnt = 0;
        }

        OCR1A = cnt;
        OCR1B = cnt;
    }
}
```

8.4 Assignment

Use the code examples from the previous paragraph to fulfil the following assignment.

Control the brightness of an LED connected to PB3 with a potentiometer.

SMART technical specification		
#	**MoS CoW**	**Description**
T1	**M**	**Connect a potentiometer in the range of 1 kΩ to 100 kΩ to ADC0.**
T2	**M**	**The LED breakout board is connected to PB0-PB5, however, only the LED connected to PB3 is used.** *TIP. Check the datasheet for the appropriate output compare channel and timer.*
T3	**M**	**The generated PWM signal must have a frequency as close as possible to 1000 Hz.** *Add a calculation in your code comments.*

SMART functional specification		
#	**MoS CoW**	**Description**
F1	**M**	**The potentiometer reading is used to control the LED brightness linearly** $\boldsymbol{PWM_{duty\ cyle}} = \frac{\boldsymbol{ADC\ value}}{\mathbf{1023}} \times \mathbf{100\%}$

TIP: If the TOP value of the timer/counter is equal to 256, then the output compare value is calculated as follows: $OCRnx = \frac{ADC\ conversion\ value}{4}$*, because the ADC value is in the range of 0 – 1023.*

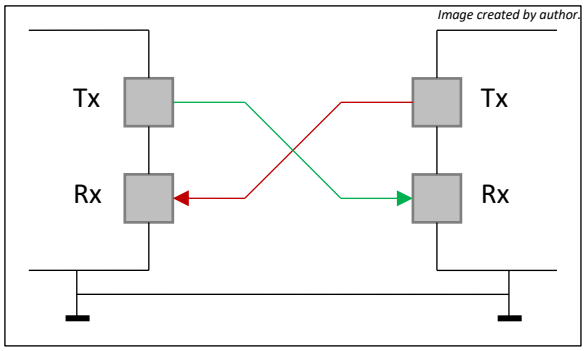
9 Week 5. Communication

9.1 Introduction

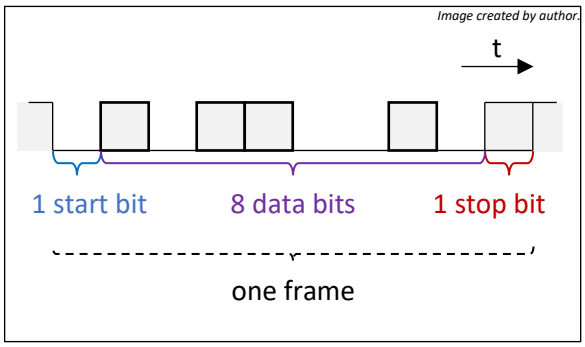
This section discusses serial communication. The ATmega328p Xplained mini board has a built-in serial to USB convertor, so no additional hardware is required for these examples to work.

9.2 Theory

There exists a variety of standards for communication between devices. A well-known standard is USB. Another standard is RS232, which describes the serial communication between two devices. Most microcontrollers support this standard with a peripheral called universal synchronous asynchronous receiver transmitter (USART). Most applications only use the asynchronous mode of operation, which means that there is no synchronisation signal (such as a clock pulse) between sender and receiver. The receiver synchronizes on the data itself.



The minimum hardware connection for sending and receiving by two devices is depicted in the adjacent image. Tx means *transmit* and Rx means *receive*.



Data is transmitted in a so-called frame. A typical example frame is depicted in the adjacent image. The idle state of the line is logic one. A frame is then started by the sender by making the line logic 0 for the duration of one bit. This is called the start bit. Then the data bits are transferred. Finally, the sender stops the transmission by making the line logic 1 for the duration of one bit. This is called the stop bit. The example clearly shows that sending one byte actually takes the duration of 10 bits.

Sender and receiver must agree on the following protocol settings:

- The speed at which bits are transmitted, a.k.a. the Baud rate or bit rate.
- The number of data bits transmitted in one frame.
- Whether parity[2] checking is used or not.
  Notice that a parity bit is not used in the example frame, so there is no parity bit.
- The number of stop bits transmitted in one frame.

An embedded software engineer does not have to toggle a digital I/O pin in software to get the result as depicted in the image. A peripheral called UART can be used for this purpose. The principle of operation of a UART is depicted in the following block diagram. The embedded software engineer configures the UART by writing the bits of configuration registers (not shown in the block diagram), for instance to set the Baud rate. Data is then automatically transmitted as soon as data is written into the transmit register. The UART peripheral automatically adds a start bit, uses a shift register to transfer the data bit-by-bit, then an optional parity bit is automatically generated, and finally the stop bit(s).

[2] https://en.wikipedia.org/wiki/Serial_port#Parity

Data is automatically received as soon as a start bit is detected by the UART peripheral. The data is received in a shift register and when all data is present, the data is moved to the receive register that can be read by software for further processing.

The bits in a status register (not shown in the block diagram) allow the software engineer to check the status of the UART, for instance if the data has been transmitted, if new data has been received, or if there was a frame error upon data reception

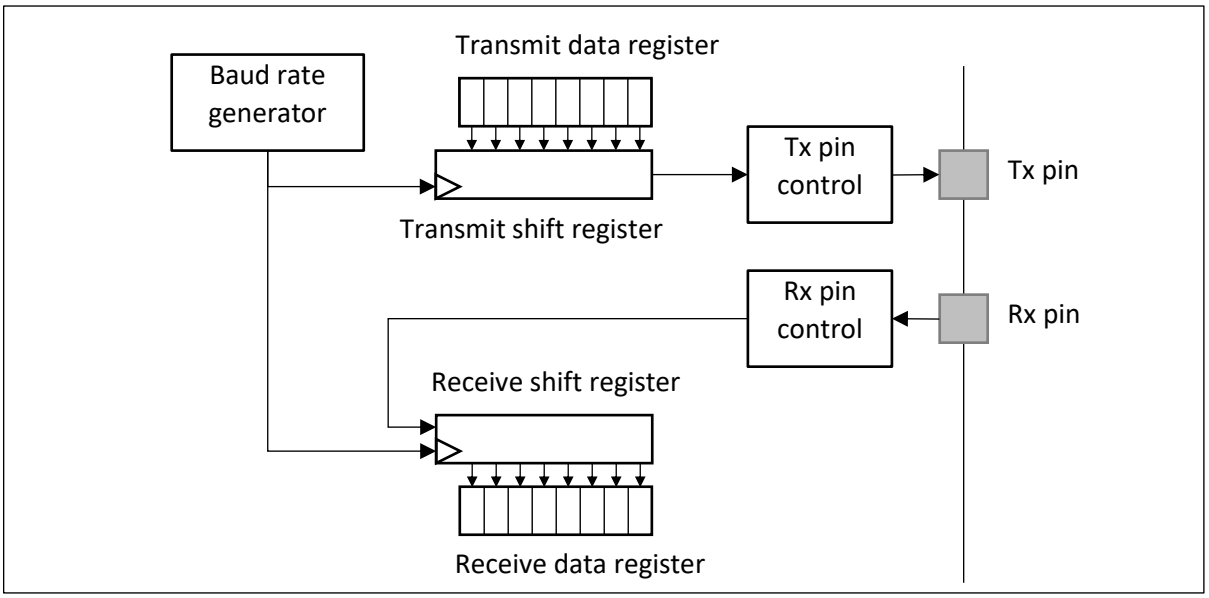


Image created by author.

9.3 Code examples

The ATmega328p microcontroller features one USART called “USART0”. The following code examples will show how to use the USART0 registers to configure the basic communication settings. However, first the installation of an additional application is required in order to visualize the data on your laptop.

9.3.1 Running the code examples

The following block diagram shows the signal path when transmitting and receiving data.

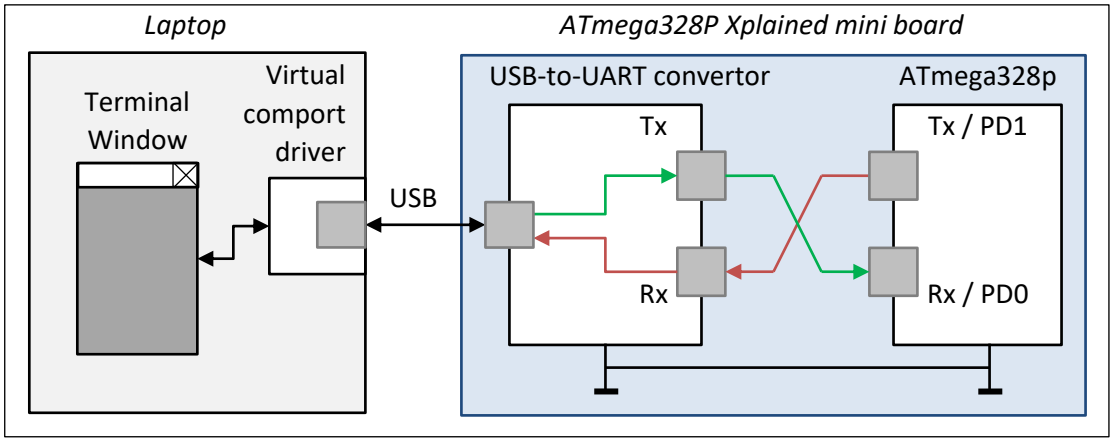


Image created by author.

Terminal Window is an application running on your laptop that is able to open a communications port. It uses a virtual comport driver (which is automatically installed the first time the ATmega328p

Xplained mini board is connected to your laptop). Data is then transferred via USB to the USB-to-UART convertor that is available on the ATmega328p Xplained mini board.

An open source terminal emulator that you can install on your laptop is called Tera Term. Another terminal emulator that you are already familiar with, is the Arduino serial monitor.

9.3.2 Communication settings

A common setting for USART communication is asynchronous mode, a baud rate of 9600 bits per second (bps), 8 data bits, no parity and 1 stop bit. This is often abbreviated by the phrase "9600-8N1". This setting will be used throughout the code examples. The datasheet (Datasheet, 2015) shows that the following registers are associated with a USART:

- UDRn
- UCSRnA
- UCSRnB
- UCSRnC
- UBRRn (which is actually a combination of the two registers UBRRnH and UBRRnL)

Notice the 'n' in this description. It must be replaced by the number of the USART you are using. As the ATmeage328p only feature one single USART, we must always replace the 'n' by 0.

Refer to section 19.10 in the datasheet (Datasheet, 2015) for a description of the registers.

9.3.2.1 Stop bit select

The number of stop bits is configured as follows:

- The number of stop bits is configured in register: **UCSRnC**
- The bit is called: **USBSn**
- The default value of this bit is: **0**
- This default value means: **1-bit is the number of stop bits**

This means that the default value of the USBSn bit is the configuration required for "9600-8N1". If you would like to set this value anyway, use the following instruction:

```
UCSR0C &= ~(1<<USBS0);
```

9.3.2.2 Parity mode

The parity mode is configured as follows:

- Parity mode is configured in register: **UCSRnC**
- The bits are called: **UPMn[1:0]**
- The default value of these bits is: **00**
- This default value means: **Disabled**

This means that the default value of the UPMn bits is the configuration required for "9600-8N1". If you would like to set this value anyway, use the following instruction:

```
UCSR0C &= ~((1<<UPM01) | (1<<UPM00));
```

9.3.2.3 Data bits

The number of data bits is configured as follows:

- The number of data bits is configured in two registers: **UCSRnC** and **UCSRnB**
- The bits are called: **UCSZn[2:0]**

- The default value of these bits is: **011**
- This default value means: **8-bit character size**

This means that the default value of the UCSZn bits is the configuration required for “9600-8N1”. If you would like to set this value anyway, use the following instructions:

```
UCSR0B &= ~(1<<UCSZ02);
UCSR0C |= (1<<UCSZ01) | (1<<UCSZ00);
```

9.3.2.4 Baud rate

It is not possible to simply write the desired baud rate, such as 9600 bps, in a register. The datasheet, however, provides equations for calculating this value, which must be written in the UBRRn register. These equations are shown in the following table:

Table 19-1. Equations for Calculating Baud Rate Register Setting

Operating Mode	Equation for Calculating Baud Rate(1)	Equation for Calculating UBRRn Value
Asynchronous normal mode (U2Xn = 0)	$BAUD = \frac{f_{OSC}}{16(UBRRn + 1)}$	$UBRRn = \frac{f_{OSC}}{16BAUD} - 1$
Asynchronous double speed mode (U2Xn = 1)	$BAUD = \frac{f_{OSC}}{8(UBRRn + 1)}$	$UBRRn = \frac{f_{OSC}}{8BAUD} - 1$
Synchronous master mode	$BAUD = \frac{f_{OSC}}{8(UBRRn + 1)}$	$UBRRn = \frac{f_{OSC}}{2BAUD} - 1$

Note: 1. The baud rate is defined to be the transfer rate in bit per second (bps)

BAUD Baud rate (in bits per second, bps)

f_{osc} System oscillator clock frequency

UBRRn Contents of the UBRRnH and UBRRnL registers, (0-4095)

The table shows that there are two options in asynchronous mode, depending on the value of the U2Xn bit. Refer to a description of this bit to section 19.3.2 in the datasheet (Datasheet, 2015). For now we will leave this U2Xn bit at its reset value, which is 0.

Calculating the UBRRn value for 9600 bps baud rate is then done as follows:

$$UBRRn = \frac{F_{osc}}{16BAUD} - 1 = \frac{F_{CPU}}{16 \times 9600} - 1 = \frac{16MHz}{16 \times 9600} - 1 = \frac{619}{6} \approx 103.167$$

The value written in UBRRn must be an integer, so rounding gives a value of 103.

```
UBRR0 = 103;
```

Instead of using a hard coded value for UBRRn, the following instruction is also often implemented:

```
UBRR0 = (uint16_t)((F_CPU / 16 / 9600) - 1);
```

This concludes the configuration of the USART.

9.3.3 Polling

Serial communication is relatively slow. When transmitting, for example, it takes time to shift the data out of the shift register. A bit called UDREn in the UCSRnA register can be polled to find out if the data register is empty, so the application can safely write the next data to be transmitted in the transmit data register. Similarly, when receiving data, the application should check if new data has

been received. This is done by polling the RXCn bit in the UCSRnA register. An implementation example is given.

```
#define F_CPU (16000000UL)

#include <avr/io.h>
#include <util/delay.h>

void usart0_init(void)
{
    // Configure USART0 in “9600-8N1”
    //
    // For this configuration, several bits can keep their reset value.
    // This means that this function only implements instructions for bits
    // that must be updated.
    // Refer to the datasheet for a description of all the bits.

    // Configure the baud rate
    UBRR0 = (uint16_t)((F_CPU / 16UL / 9600) - 1);

    // Receiver and transmitter enable
    UCSR0B |= (1<<RXEN0) | (1<<TXEN0);
}

char usart0_receive(void)
{
    // Wait for data to be received
    while( !(UCSR0A & (1<<RXC0)) )
    {;}

    // Get and return received data from buffer
    return UDR0;
}

void usart0_transmit(char data)
{
    // Wait for empty transmit buffer
    while( !(UCSR0A & (1<<UDRE0)) )
    {;}

    // Put data into buffer, sends the data
    UDR0 = data;
}

void usart0_transmitStr(char *str)
{
    while(*str)
    {
        usart0_transmit(*str++);
    }
}

int main(void)
{
    // PB0 - PB5 output, rest input
    DDRB = 0b00111111;

    usart0_init();
    usart0_transmitStr("MIC2 - USART with polling\r\n");

    while (1)
    {
```

```
        // Toggle pin PB0
        PORTB ^= (1<<PORTB0);

        // Wait for incoming data
        char data = usart0_receive();

        // Check the data
        if(data == '1')
        {
            // LED on
            PORTB |= (1<<PORTB1);

            usart0_transmitStr("On\r\n");
        }
        else
        {
            // LED off
            PORTB &= ~(1<<PORTB1);

            usart0_transmitStr("Off\r\n");
        }
    }
}
```

9.3.4 Interrupts

One major disadvantage of using software polling is that the microcontroller spends time executing code (the while-statements in the previous examples). All this time, no other code can be executed. If there is for example no incoming data, the application would wait indefinitely. So instead of polling, the microcontroller can trigger an ISR as soon as data has been received or transmitted. The basic idea is then to create two buffers, one for transmitting and one for receiving. The transmit buffer is written as fast as possible in the main application and read byte-by-byte as soon as the USART is ready to transmit new data. The receive buffer is written by the ISR as soon as new data is received and read by the main application as soon as the main application requires the data.

```
/**
* \file
*
* \brief USART Interrupt example
*
* Copyright (C) 2016 Atmel Corporation. All rights reserved.
* Modified by H. Arends for HAN University of Applied Sciences to support
* the ATmega328P microcontroller.
*
* \page License
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions are met:
*
* 1. Redistributions of source code must retain the above copyright notice,
* this list of conditions and the following disclaimer.
*
* 2. Redistributions in binary form must reproduce the above copyright notice,
* this list of conditions and the following disclaimer in the documentation
* and/or other materials provided with the distribution.
*
* 3. The name of Atmel may not be used to endorse or promote products derived
* from this software without specific prior written permission.
*
* 4. This software may only be redistributed and used in connection with an
```

```
* Atmel microcontroller product.
*
* THIS SOFTWARE IS PROVIDED BY ATMEL "AS IS" AND ANY EXPRESS OR IMPLIED
* WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
* MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT ARE
* EXPRESSLY AND SPECIFICALLY DISCLAIMED. IN NO EVENT SHALL ATMEL BE LIABLE FOR
* ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
* ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
* POSSIBILITY OF SUCH DAMAGE.
*
*/
/*
* Support and FAQ: visit
* <a href="http://www.atmel.com/design-support/">Atmel Support</a>
*/
#include "avr/io.h"
#include "avr/interrupt.h"

#define F_CPU 16000000UL
#include "util/delay.h"

// UART Buffer Defines
// Supported sizes: 2,4,8,16,32,64,128 or 256 bytes
#define USART_RX_BUFFER_SIZE (64)
#define USART_TX_BUFFER_SIZE (64)

// Buffer size checks
#define USART_RX_BUFFER_MASK (USART_RX_BUFFER_SIZE - 1)
#if (USART_RX_BUFFER_SIZE & USART_RX_BUFFER_MASK)
    #error RX buffer size is not a power of 2
#endif

#define USART_TX_BUFFER_MASK (USART_TX_BUFFER_SIZE - 1)
#if (USART_TX_BUFFER_SIZE & USART_TX_BUFFER_MASK)
    #error TX buffer size is not a power of 2
#endif

// Static variables
static char USART_RxBuf[USART_RX_BUFFER_SIZE];
static volatile unsigned char USART_RxHead;
static volatile unsigned char USART_RxTail;

static char USART_TxBuf[USART_TX_BUFFER_SIZE];
static volatile unsigned char USART_TxHead;
static volatile unsigned char USART_TxTail;

// Function prototypes
void usart0_init(void);
char usart0_receive(void);
void usart0_transmit(char data);
unsigned char usart0_nUnread(void);
void usart0_transmitStr(char *str);

int main(void)
{
    // PB0 - PB5 output, rest input
    DDRB = 0b00111111;
```

```
    // Initialize the USART
    usart0_init();
    usart0_transmitStr("MIC2 - USART with interrupts\r\n");

    sei();

    while(1)
    {
        _delay_ms(100);

        // Toggle pin PB0
        PORTB ^= (1<<PORTB0);

        // Check for unread bytes in the receive buffer
        unsigned char nBytes = usart0_nUnread();

        if(nBytes > 0)
        {
            char data = usart0_receive();

            // Check the data
            if(data == '1')
            {
                // LED on
                PORTB |= (1<<PORTB1);

                usart0_transmitStr("On\r\n");
            }
            else
            {
                // LED off
                PORTB &= ~(1<<PORTB1);

                usart0_transmitStr("Off\r\n");
            }
        }
    }

    return 0;
}

/*
 * Initialize the USART.
 */
void usart0_init(void)
{
    // Configure the baud rate
    UBRR0 = (uint16_t)((F_CPU / 16UL / 9600) - 1);

    // Enable USART receiver and transmitter
    UCSR0B = ((1<<RXEN0) | (1<<TXEN0) | (1<<RXCIE0));

    // Flush buffers
    unsigned char x = 0;
    USART_RxTail = x;
    USART_RxHead = x;
    USART_TxTail = x;
    USART_TxHead = x;
}

/*
 * Interrupt handler for received data.
```

```
 * Data is placed in the receive buffer.
 */
ISR(USART_RX_vect)
{
    char data;
    unsigned char tmphead;

    // Read the received data
    data = UDR0;

    // Calculate buffer index
    tmphead = (USART_RxHead + 1) & USART_RX_BUFFER_MASK;

    // Store new index
    USART_RxHead = tmphead;

    if (tmphead == USART_RxTail)
    {
        // ERROR! Receive buffer overflow
    }

    // Store received data in buffer
    USART_RxBuf[tmphead] = data;
}

/*
 * Interrupt handler for transmit data.
 * Data is read from the transmit buffer. If all data was transmitted,
 * transmit interrupts are disabled.
 */
ISR(USART_UDRE_vect)
{
    unsigned char tmptail;

    // Check if all data is transmitted
    if (USART_TxHead != USART_TxTail)
    {
        // Calculate buffer index
        tmptail = ( USART_TxTail + 1 ) & USART_TX_BUFFER_MASK;

        // Store new index
        USART_TxTail = tmptail;

        // Start transmission
        UDR0 = USART_TxBuf[tmptail];
    }
    else
    {
        // Disable UDRE interrupt
        UCSR0B &= ~(1<<UDRIE0);
    }
}

/*
 * This function returns a new byte from the receive buffer.
 * It will wait for data to be available! Use the function
 * usart0_nUnreadBytes() to make sure data is available.
 */
char usart0_receive(void)
{
    unsigned char tmptail;
```

```
    // Wait for incoming data
    while (USART_RxHead == USART_RxTail);

    // Calculate buffer index
    tmptail = (USART_RxTail + 1) & USART_RX_BUFFER_MASK;

    // Store new index
    USART_RxTail = tmptail;

    // Return data
    return USART_RxBuf[tmptail];
}

/*
 * This function places a new byte in the transmit buffer
 * and starts a new transmission by enabling transmit interrupt.
 */
void usart0_transmit(char data)
{
    unsigned char tmphead;

    // Calculate buffer index
    tmphead = (USART_TxHead + 1) & USART_TX_BUFFER_MASK;

    // Wait for free space in buffer
    while (tmphead == USART_TxTail);

    // Store data in buffer
    USART_TxBuf[tmphead] = data;

    // Store new index
    USART_TxHead = tmphead;

    // Enable UDRE interrupt
    UCSR0B |= (1<<UDRIE0);
}

/*
 * This function returns the number of unread bytes in the receive buffer.
 */
unsigned char usart0_nUnread(void)
{
    if(USART_RxHead == USART_RxTail)
        return 0;
    else if(USART_RxHead > USART_RxTail)
        return USART_RxHead - USART_RxTail;
    else
        return USART_RX_BUFFER_SIZE - USART_RxTail + USART_RxHead;
}

/*
 * Transmits a string of characters to the USART.
 * The string must be terminated with '\0'.
 *
 * - This function uses the function uart0_transmit() to
 *   transmit a byte via the USART
 * - Bytes are transmitted until the terminator
 *   character '\0' is detected. Then the function returns.
 */
void usart0_transmitStr(char *str)
{
    while(*str)
```

```
    {
        usart0_transmit(*str++);
    }
}
```

9.4 Assignment

Use the code examples from the previous paragraph to fulfil the following assignment.

Implement a code lock. The user gets three tries. If the third successive try is invalid, the code lock is blocked, until the reset button is clicked.

SMART technical specification		
#	**MoS CoW**	**Description**
T1	**M**	**The LED breakout board is connected to PB0-PB5.**
T2	**M**	**Use the switch that is already available on the ATmega328p Xplained mini board.**

SMART functional specification		
#	**MoS CoW**	**Description**
F1	**M**	**Print a welcome message to the Terminal Window application as soon as the application starts. The welcome message shows the following information:**
F1.1	M	Your name
F1.2	M	Your student number
F1.3	M	*The following phrase: "Week 5. Communication"*
F2	**M**	**The application contains a secret code that is equal to your student number.**
F3	**M**	**When the application starts:**
F3.1	M	The LED connected to PB5 is off.
F3.2	M	The LEDs connected to PB0, PB1 and PB2 are off.
F3.3	M	Use the Terminal Window application to ask the user to input the secret code.
F4	**M**	**A code is entered by the user in the Arduino Serial Monitor.**
F4.1	M	The entered code is a string of characters that must be terminated by a Newline ('\n') character. *TIP. Wait until the number of received bytes is equal to the length of your secret code + 1, and then process all the data.*
F4.2	M	If the string doesn't exactly have the same number of characters as the secret code (excluding the Newline character), the code is considered invalid.
F4.3	M	If the entered code is not equal to your student number, the entered code is considered invalid.
F4.4	M	If the entered code is valid:
F4.4.1	M	The user gets three new tries.
F4.4.2	M	The LEDs connected to PB0, PB1 and PB2 turn off.
F4.4.3	M	A message in the Terminal Window application tells that the valid code was entered and also the number of tries.
F5	**M**	**Three successive invalid entered codes will lock the application.**
F5.1	M	After the first invalid entered code, the LED connected to PB0 turns on. The user has two more tries.
F5.2	M	After the second successive invalid entered code, the LED connected to PB1 also turns on. The user has one more tries.

SMART functional specification		
#	**MoS CoW**	**Description**
F5.3	M	After the third successive invalid entered code, the LED connected to PB2 also turns on. The user has no more tries. A message telling that the application is locked is shown on the Terminal Window application. The application does not accept any codes anymore, until the application is reset (see next requirement).
F6	**M**	**The application can be reset by pressing the switch.**
F6.1	M	The user gets three new tries.
F6.2	M	The LEDs connected to PB0, PB1 and PB2 turn off.
F6.3	M	A message in the Terminal Window application tells that the code lock has been reset.

10 Week 6. Sensors and Actuators

10.1 Introduction

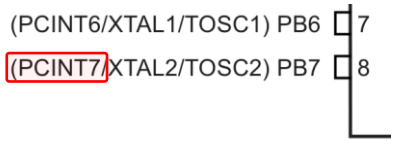
This section demonstrates how to use the peripherals for reading sensors and writing actuators. The theory section starts with an introduction to pin change interrupts and software design.

10.2 Theory

10.2.1 Pin change interrupts

The value of a digital input pin can be read from the PINx register (see Week 2. Digital - Input pins). However, instead of polling the value of the PINx bit, the ATmega328p microcontroller also features hardware that can generate an interrupt as soon as an pin change is detected. This hardware peripheral is call Pin Change Interrupts.

Each pin of the microcontroller is assigned an unique PCINT number. PB7, for instance, is assigned the number PCINT7, as mentioned in the pin configurations section of the datasheet (Datasheet, 2015):



The ATmega328p microcontroller combines multiple PCINT interrupts into one interrupt request and hence one ISR. This is depicted in the following tables:

PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16
PCINT Interrupt request **2**							

```
ISR(PCINT2_vect)
{

}
```

	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8
PCINT Interrupt request **1**							

```
ISR(PCINT1_vect)
{

}
```

PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0
PCINT Interrupt request **0**							

```
ISR(PCINT0_vect)
{

}
```

Enabling/disabling each PCINT Interrupt request [2:0] is done in register PCICR.
Enabling/disabling each PCINT[0:7] is done register PCMSK0.
Enabling/disabling each PCINT[8:14] is done register PCMSK1.
Enabling/disabling each PCINT[23:16] is done register PCMSK2.

If multiple PCINTs are enable for one PCINT interrupt request, the code in the ISR should check which of the pins triggered the ISR.

This code demonstrates how to configure pin change interrupts for PB7 and toggle the LED connected to PB5.

```
#define F_CPU (16000000UL)

#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdbool.h>

volatile bool flag = false;
```

```
// This is the Interrupt Service Routine (ISR). This 'function' is executed
// every time a pin change has occurred, because we configured that way in
// main().
// You cannot choose the name of an ISR. These are predefined in the header
// file called iom328p.h.
ISR(PCINT0_vect)
{
    // When this functions is executed, the pin has changed, either from
    // high-to-low (falling edge) or from low-to-high (rising edge).

    // Pressing the switch generates a high-to-low transition. So if we would
    // like to know if the switch was pressed, the pin value should read
    // 0 now.
    if((PINB & (1<<PINB7)) == 0)
    {
        // Yes, switch was pressed. Set the global flag so we can handle the
        // event in main().
        flag = true;
    }
}

int main(void)
{
    // PB0 - PB5 output, rest input
    DDRB = 0b00111111;

    // Configure pin change interrupts for PB7
    //
    // - PB7 is connected to PCINT7, we must therefore enable the PCINT7
    //   bit in the PCMSK0 register
    // - PCINT7 is part of group 0 pin change interrupts, we must therefore
    //   enable the PCIE0 bit in the PCICR register
    PCMSK0 |= (1<<PCINT7);
    PCICR  |= (1<<PCIE0);

    sei();

    while (1)
    {
        // Check the flag
        if(flag == true)
        {
            // Reset the flag
            flag = false;

            // Toggle the LED
            PINB = (1<<PINB5);
        }
    }
}
```

10.2.2 Software design

In order to keep the code as short as possible in an ISR, we need a means of communication between an ISR and the main program. This is possible by using *global variables*. The ISR updates one or more global variables and the main program handles the updated information when possible.

The following example shows the design of a simple application that is able to handle switch presses and timer expiration events simultaneously in the eyes of a user.

First declare global variables. These global variables are referred to as *flags*, because they signal the main loop that an hardware event has been triggered. Make sure to declare these global variable **volatile**. This keyword prevents optimizations by the compiler.

```
// The keyword volatile ensures that the compiler will not try to
// do any optimizations on this variable.
volatile unsigned char flag_switch_pressed = 0;
volatile unsigned char flag_5s_expired = 0;
volatile unsigned char cnt = 0;
```

Every time the switch is pressed, the following ISR is executed. Only when a falling edge is detected, which means the key is pressed (and not released), the value of the flag is set to 1.

```
ISR(PCINT0_vect)
{
    // A transition (high-to-low (falling) or low-to-high (rising)) is detected.
    // If the current value of the pin is low, it must have been a falling edge.
    if((PINB & (1<<PINB7)) == 0)
    {
        // Set flag
        flag_switch_pressed = 1;
    }
}
```

Every time a timer/counter 1 interrupt is generated, the following ISR is executed. In the main program, the ISR will be initialized to trigger every 500ms. When the ISR is triggered every tenth successive time, the value of the flag is set to 1.

```
ISR(TIMER1_COMPA_vect)
{
    cnt++;

    // Is the ISR called 10 times?
    if(cnt >= 10)
    {
        // Yes, reset count and set the flag
        cnt = 0;
        flag_5s_expired = 1;
    }
}
```

Finally, the main program initializes the peripherals and processes the events. Notice how if-statements are used for checking the flags. As soon as an if-statement is true, which means the flag has been set, the flag is reset immediately. A global variable of type state_t is used to keep track of the system state, so we know how to handle an event in a specific state. Due to this design, the microcontroller is mainly busy checking the if-statements, because most of the time these will evaluate to false.

```
typedef enum
{
    S_IDLE,
    S_LED_ON,
    S_LED_OFF,
}state_t;

volatile state_t state = S_IDLE;
```

```
int main(void)
{
    // PB0 to PB5 output, PB6 and PB7 input
    DDRB = 0b00111111;

    // PCINT7 enable
    PCMSK0 |= (1<<PCINT7);
    PCICR  |= (1<<PCIE0);

    // Initialize 16-bit Timer 1:
    // - CTC mode of operation with TOP in OCR1A
    // - 256 prescaler
    // - Set output compare value for channel A for generating an
    //   interrupt every 500ms
    //   TOP = (F_CPU / N / 2 Hz) - 1 = (16MHz / 256 / 2Hz) - 1 = 31249
    //   31249 < 2^16, so it will fit in 16-bit register OCR1A
    // - Enable output compare match interrupt for channel A
    OCR1A = 31249;
    TIMSK1 |= (1<<OCIE1A);
    TCCR1B |= (1<<WGM12);

    sei();

    state = S_LED_OFF;

    while(1)
    {
        // Switch pressed?
        if(flag_switch_pressed == 1)
        {
            // Reset the flag
            flag_switch_pressed = 0;

            // Check the current state and take the appropriate action
            switch(state)
            {
            case S_LED_ON: // Fall through
            case S_LED_OFF:
            {
                // LED on
                PORTB |= (1<<PORTB5);

                // Reset and start the timer
                cnt = 0;
                TCNT1 = 0;
                TCCR1B |= (1<<CS12);

                // Next state
                state = S_LED_ON;
            }
            break;
            default:
            {
                ; // Event unexpected in this state, do nothing
            }
            break;
            }
        }

        // 5 seconds expired?
        if(flag_5s_expired == 1)
```

```
        {
            // Reset the flag
            flag_5s_expired = 0;

            // Check the current state and take the appropriate action
            switch(state)
            {
            case S_LED_ON:
            {
                // LED off
                PORTB &= ~(1<<PORTB5);

                // Stop the timer
                TCCR1B &= ~(1<<CS12);

                // Next state
                state = S_LED_OFF;
            }
            break;
            case S_LED_OFF: // Fall through
            default:
            {
                ; // Event unexpected in this state, do nothing
            }
            break;
            }
        }
    }
}
```

10.3 Code examples

This example shows the step-by-step implementation of an interrupt driven system that reads distance information from the HC-SR04 ultrasonic range finder and displays the distance in centimetres on an LCD. The display will also show the value of a switch being pressed or not. The USART will be used for displaying debug information.

First, we consider the pins that are required to implement the functionality:

- The USART requires two dedicated pins: PD0 and PD1.
- The switch requires one dedicated pin: PB7.
- The LCD requires six digital output pins. We choose PD2 to PD7.
- The HC-SR04 requires one digital input and one digital output pin. We choose PB0 and PB1.

To keep the project nice and clear, we create a project that has a file for each dedicated part. The body of the main is initially implemented as follows:

main.c

```
#define F_CPU (16000000UL)

#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdbool.h>

void setup(void)
{

}
```

```
void loop(void)
{

}

int main(void)
{
    setup();

    // Global interrupts enable
    sei();

    while (1)
    {
        loop();
    }
}
```

The setup() function is called one time and is intended for calling configuration functions. Then, in an endless loop, the loop() function is called as often as possible. The loop() function implements the system's functionality. Notice that ISR's will also be executed and that the loop() function will be notified by flags!

10.3.1 Switch

Let's start by implementing the behaviour for the switch on the ATmega328p Xplained mini board. This switch is called "sw0". We will therefore add two files to the project called sw0.h and sw0.c. These files are implemented as follows:

sw0.h

```
#ifndef SW0_H_
#define SW0_H_

#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdbool.h>

void sw0_init(void);
bool sw0_pressed(void);

#endif /* SW0_H_ */
```

sw0.c

```
#include "sw0.h"

void sw0_init(void)
{
    // Make sure PB7 is an input pin
    DDRB &= ~(1<<DDB7);
}

bool sw0_pressed(void)
{
    // Returns true or false, depending on the pin value
    return ((PINB & (1<<PINB7)) == 0);
}
```

Switch sw0 presses can then be used in the main as follows:

main.c

```
void setup(void)
{
    sw0_init();
}

void loop(void)
{
    // Is the switch pressed?
    if(sw0_pressed())
    {
        // Implement SW0 functionality here later ...
    }
}
```

10.3.2 Serial communication

The next step is to add support for serial communication via usart0. Two more files are added to the project, with code taken from *Week 5. Communication*.

usart0.h

```
#ifndef USART0_H_
#define USART0_H_

void usart0_init(void);
char usart0_receive(void);
void usart0_transmit(char data);
unsigned char usart0_nUnread(void);
void usart0_transmitStr(char *str);

#endif /* USART0_H_ */
```

usart0.c

```
/**
* \file
*
* \brief USART Interrupt example
*
* Copyright (C) 2016 Atmel Corporation. All rights reserved.
* Modified by H. Arends for HAN University of Applied Sciences to support
* the ATmega328P microcontroller.
*
* \page License
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions are met:
*
* 1. Redistributions of source code must retain the above copyright notice,
* this list of conditions and the following disclaimer.
*
* 2. Redistributions in binary form must reproduce the above copyright notice,
* this list of conditions and the following disclaimer in the documentation
* and/or other materials provided with the distribution.
*
* 3. The name of Atmel may not be used to endorse or promote products derived
* from this software without specific prior written permission.
*
* 4. This software may only be redistributed and used in connection with an
```

```
* Atmel microcontroller product.
*
* THIS SOFTWARE IS PROVIDED BY ATMEL "AS IS" AND ANY EXPRESS OR IMPLIED
* WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
* MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT ARE
* EXPRESSLY AND SPECIFICALLY DISCLAIMED. IN NO EVENT SHALL ATMEL BE LIABLE FOR
* ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
* ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
* POSSIBILITY OF SUCH DAMAGE.
*
*/
/*
* Support and FAQ: visit
* <a href="http://www.atmel.com/design-support/">Atmel Support</a>
*/
#define F_CPU 16000000UL

#include "avr/io.h"
#include "avr/interrupt.h"
#include "usart0.h"

// UART Buffer Defines
// Supported sizes: 2,4,8,16,32,64,128 or 256 bytes
#define USART_RX_BUFFER_SIZE (64)
#define USART_TX_BUFFER_SIZE (64)

// Buffer size checks
#define USART_RX_BUFFER_MASK (USART_RX_BUFFER_SIZE - 1)
#if (USART_RX_BUFFER_SIZE & USART_RX_BUFFER_MASK)
    #error RX buffer size is not a power of 2
#endif

#define USART_TX_BUFFER_MASK (USART_TX_BUFFER_SIZE - 1)
#if (USART_TX_BUFFER_SIZE & USART_TX_BUFFER_MASK)
    #error TX buffer size is not a power of 2
#endif

// Static variables
static char USART_RxBuf[USART_RX_BUFFER_SIZE];
static volatile unsigned char USART_RxHead;
static volatile unsigned char USART_RxTail;

static char USART_TxBuf[USART_TX_BUFFER_SIZE];
static volatile unsigned char USART_TxHead;
static volatile unsigned char USART_TxTail;

/*
 * Initialize the USART.
 */
void usart0_init(void)
{
    // Configure the baud rate
    UBRR0 = (uint16_t)((F_CPU / 16UL / 9600) - 1);

    // Enable USART receiver and transmitter
    UCSR0B = ((1<<RXEN0) | (1<<TXEN0) | (1<<RXCIE0));

    // Flush buffers
```

```
    unsigned char x = 0;
    USART_RxTail = x;
    USART_RxHead = x;
    USART_TxTail = x;
    USART_TxHead = x;
}

/*
 * Interrupt handler for received data.
 * Data is placed in the receive buffer.
 */
ISR(USART_RX_vect)
{
    char data;
    unsigned char tmphead;

    // Read the received data
    data = UDR0;

    // Calculate buffer index
    tmphead = (USART_RxHead + 1) & USART_RX_BUFFER_MASK;

    // Store new index
    USART_RxHead = tmphead;

    if (tmphead == USART_RxTail)
    {
        // ERROR! Receive buffer overflow
    }

    // Store received data in buffer
    USART_RxBuf[tmphead] = data;
}

/*
 * Interrupt handler for transmit data.
 * Data is read from the transmit buffer. If all data was transmitted,
 * transmit interrupts are disabled.
 */
ISR(USART_UDRE_vect)
{
    unsigned char tmptail;

    // Check if all data is transmitted
    if (USART_TxHead != USART_TxTail)
    {
        // Calculate buffer index
        tmptail = ( USART_TxTail + 1 ) & USART_TX_BUFFER_MASK;

        // Store new index
        USART_TxTail = tmptail;

        // Start transmission
        UDR0 = USART_TxBuf[tmptail];
    }
    else
    {
        // Disable UDRE interrupt
        UCSR0B &= ~(1<<UDRIE0);
    }
}
```

```c
/*
 * This function returns a new byte from the receive buffer.
 * It will wait for data to be available! Use the function
 * usart0_nUnreadBytes() to make sure data is available.
 */
char usart0_receive(void)
{
    unsigned char tmptail;

    // Wait for incoming data
    while (USART_RxHead == USART_RxTail);

    // Calculate buffer index
    tmptail = (USART_RxTail + 1) & USART_RX_BUFFER_MASK;

    // Store new index
    USART_RxTail = tmptail;

    // Return data
    return USART_RxBuf[tmptail];
}

/*
 * This function places a new byte in the transmit buffer
 * and starts a new transmission by enabling transmit interrupt.
 */
void usart0_transmit(char data)
{
    unsigned char tmphead;

    // Calculate buffer index
    tmphead = (USART_TxHead + 1) & USART_TX_BUFFER_MASK;

    // Wait for free space in buffer
    while (tmphead == USART_TxTail);

    // Store data in buffer
    USART_TxBuf[tmphead] = data;

    // Store new index
    USART_TxHead = tmphead;

    // Enable UDRE interrupt
    UCSR0B |= (1<<UDRIE0);
}

/*
 * This function returns the number of unread bytes in the receive buffer.
 */
unsigned char usart0_nUnread(void)
{
    if(USART_RxHead == USART_RxTail)
        return 0;
    else if(USART_RxHead > USART_RxTail)
        return USART_RxHead - USART_RxTail;
    else
        return USART_RX_BUFFER_SIZE - USART_RxTail + USART_RxHead;
}

/*
 * Transmits a string of characters to the USART.
 * The string must be terminated with '\0'.
```

```
 *
 * - This function uses the function uart0_transmit() to
 *   transmit a byte via the USART
 * - Bytes are transmitted until the terminator
 *   character '\0' is detected. Then the function returns.
 */
void usart0_transmitStr(char *str)
{
    while(*str)
    {
        usart0_transmit(*str++);
    }
}
```

Serial communication is tested by transmitting a string in the loop() function when SW0 is pressed:

main.c

```
void setup(void)
{
    sw0_init();
    usart0_init();
}

void loop(void)
{
    // Is the switch pressed?
    if(sw0_pressed())
    {
        usart0_transmitStr("SW0 pressed\r\n");
    }
}
```

10.3.3 LCD

An LCD is a module that integrates a display controller and liquid crystal segments. The display controller is a microcontroller that can be instructed what to show on the liquid crystal segments through a parallel digital interface. A parallel interface that is often used in display controllers is called HD44780 and requires a minimum of 6 digital pins. The following example implements an LCD driver for the ATmega328p microcontroller.

lcd.h

```
#ifndef LCD_H_
#define LCD_H_

#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdbool.h>

void lcd_init(void);
void lcd_clear(void);
void lcd_print(char *str);
void lcd_cursor(unsigned char row, unsigned char col);

#endif /* LCD_H_ */
```

lcd.c

```
#define F_CPU 16000000UL
```

```c
#include "lcd.h"
#include "util/delay.h"

// Use defines for readability and portability
#define E  PORTD2
#define RS PORTD3

// Writes 4-bit data to LCD
void lcd_write_4bit(char data)
{
    // Set the port pins according the data
    PORTD &= ~(0xF0);
    PORTD |= (data << 4);

    // Cycle the E pin
    PORTD |= (1<<E);
    _delay_us(1);
    PORTD &= ~(1<<E);
    _delay_us(35);
}

// Write data to the LCD
void lcd_write_data(char data)
{
    PORTD |= (1<<RS);

    lcd_write_4bit(data>>4);
    lcd_write_4bit(data);
}

// Write command to the LCD
void lcd_write_command(char cmd)
{
    PORTD &= ~(1<<RS);

    lcd_write_4bit(cmd>>4);
    lcd_write_4bit(cmd);
}

// Initialize the digital pins
// Next, initialize the LCD by sending the commands as mentioned
// in the LCD's datasheet.
void lcd_init(void)
{
    // PD2 - PD7 output, others unchanged
    DDRD |= 0b11111100;

    // LCD startup delay
    _delay_us(40000);

    // Setup the LCD

    PORTD &= ~(1<<RS);

    // Send the commands. Make sure that the commands are not transmitted
    // too fast by adding a delay after every command. The delays are
    // chosen with a save margin. The minimum required delay are mentioned
    // in the LCD datasheet.
    lcd_write_4bit(0x3);
    _delay_us(4100);
    lcd_write_4bit(0x3);
    _delay_us(100);
```

```
    lcd_write_4bit(0x3);

    lcd_write_4bit(0x2);
    lcd_write_command(0x28);
    lcd_write_command(0x0C);
    lcd_write_command(0x06);
    lcd_write_command(0x01);
    _delay_us(2000);
}

// Clear the LCD
void lcd_clear(void)
{
    lcd_write_command(0x01);
    _delay_us(2000);

    lcd_cursor(0,0);
}

// Set the cursor to the position indicated by row and col
void lcd_cursor(unsigned char row, unsigned char col)
{
    char address;

    address = (row * 0x40) + col;
    address |= 0x80;
    lcd_write_command(address);
}

// Put one ASCII character on the current cursor position
void lcd_putchar(char c)
{
    lcd_write_data(c);
}

// Print a string of ASCII characters to the LCD, starting at the
// current cursor position
void lcd_print(char *str)
{
    while(*str)
    {
        lcd_putchar(*str++);
    }
}
```

Testing the LCD driver in main:

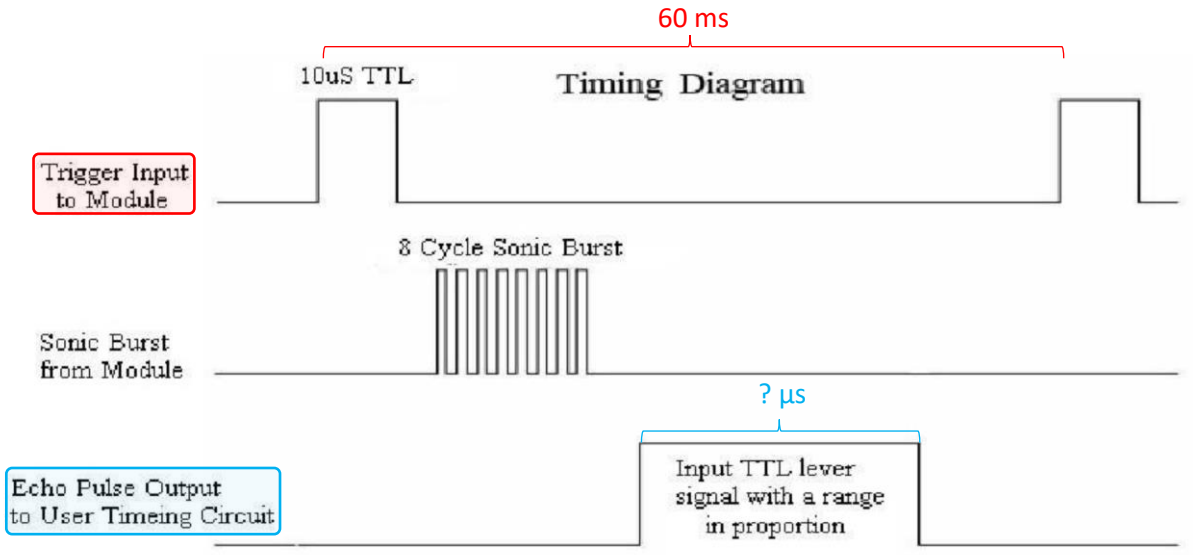
main.c

```
void setup(void)
{
    sw0_init();
    usart0_init();
    lcd_init();

    lcd_print("MIC2 example");
}

void loop(void)
{
    // Is the switch pressed?
    if(sw0_pressed())
```

```
    {
        usart0_transmitStr("SW0 pressed\r\n");

        lcd_cursor(1,0);
        lcd_print("SW0 pressed");
    }
}
```

10.3.4 HC-SR04 ultrasonic range finder

The HC-SR04 ultrasonic range finder operating principle is shown in the following timing diagram, taken from the datasheet (HC-SR04):



The signal *Trigger Input to Module* is a digital input to the HC-SR04 module and an output port pin of the ATmega328p microcontroller. The datasheet suggests that this signal should not cycle within 60ms. As this is a repeating signal, we choose to create a PWM signal by using one of the timers. To make sure that we are on the save side, the T_{PWM} will be (approximately) 100ms and the duty cycle (approximately) 10%.

The signal *Sonic Burst from Module* is generated by the module and not used by the microcontroller.

The signal *Echo Pulse Output to User Timing Circuit* is a digital output of the HC-SR04 module and an input for the ATmega328p microcontroller. The width of the pulse indicates the distance to an object. A timer/counter will be used to determine the width of this pulse by taking the difference of the timer counter value at a rising and a falling edge. This difference, together with the timer/counter counting frequency, provides the width of the pulse which can be used to calculate the distance in centimetres.

The following image shows the interaction of the signals, the timer/counter 1 and the Pin Change Interrupts.

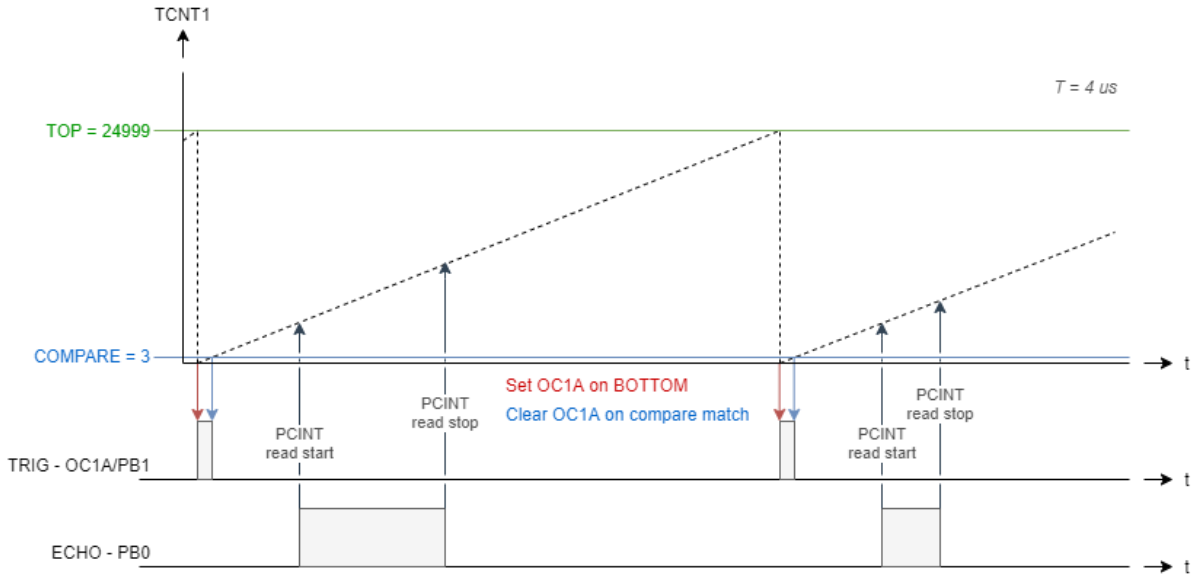


Image created by author.

The timer/counter 1 will be configured to count at a frequency of 250kHz. This means a counting pulse every 4µs. Timer/counter 1 generates a PWM signal on channel A, with a pulse that is high for 4 counts (3 + 1). This means that the TRIG pulse will be high for 16µs, which is according to the specification of at least 10µs. By setting the TOP value to 24999, this cycle will repeat every $25000 \times 4\mu s = 100ms$.

Pin change interrupts are used on the ECHO pin. On both a rising and falling edge, the timer/counter 1 count value is read from the register called TCNT1. On a falling edge, these values are used to calculate the distance as mentioned in the datasheet (HC-SR04):

$$distance\_in\_cm = \frac{(stop - start) \times 4}{58}$$

The difference between stop and start is multiplied by four, because every counting pulse takes 4µs.

The calculated *distance_in_cm* will be a global variable that is updated every 100ms and can be read in the main application.

The following files are an initial template for this sensor.

hcsr04.h

```
#ifndef HCSR04_H_
#define HCSR04_H_

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

extern volatile uint16_t hcsr04_distance_cm;

void hcsr04_init(void);

#endif /* HCSR04_H_ */
```

hcsr04.c

```
#include "hcsr04.h"
```

```
volatile uint16_t hcsr04_distance_cm = 0;

ISR(PCINT0_vect)
{
    static uint16_t timer_start = 0;
    static uint16_t timer_stop = 0;

    // To be implemented …
}

void hcsr04_init(void)
{
    // To be implemented …
}
```

A statemachine is used in the main application to accomplish the following behaviour:

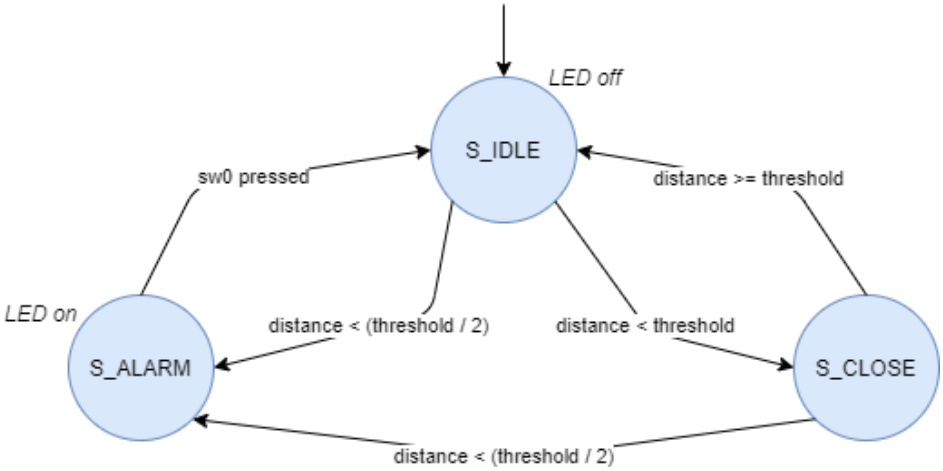


Image created by author.

main.c

```
#define F_CPU (16000000UL)

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

#include "sw0.h"
#include "usart0.h"
#include "lcd.h"
#include "hcsr04.h"
#include "adc0.h"

// System states
typedef enum
{
    S_IDLE,
    S_CLOSE,
    S_ALARM,

}state_t;

volatile state_t state;
```

```
void setup(void)
{
    // PB5 output
    DDRB |= (1<<DDB5);

    sw0_init();
    usart0_init();
    lcd_init();
    hcsr04_init();
    adc0_init();

    usart0_transmitStr("Welcome to this MIC2 demo\r\n");

    state = S_IDLE;
    usart0_transmitStr("state = S_IDLE\r\n");
}

void loop(void)
{
    // Switch pressed event?
    if(sw0_pressed())
    {
        // Check the current state and take the appropriate action
        switch(state)
        {
        case S_ALARM:
        {
            // Set the next state
            state = S_IDLE;
            usart0_transmitStr("state = S_IDLE\r\n");

            // LED off
            PORTB &= ~(1<<PORTB5);
        }
        break;
        case S_IDLE: // Fall through
        case S_CLOSE: // Fall through
        default:
        {
            ; // Event unexpected in this state, do nothing
        }
        break;
        }
    }

    // Get the threshold value
    uint16_t threshold_cm = adc0_getthreshold();

    // Is the distance below the threshold?
    if(hcsr04_distance_cm < (threshold_cm/2))
    {
        // Check the current state and take the appropriate action
        switch(state)
        {
            case S_IDLE:
            case S_CLOSE: // Fall through
            {
                // Set the next state
                state = S_ALARM;
                usart0_transmitStr("state = S_ALARM\r\n");
```

```
                // LED on
                PORTB |= (1<<PORTB5);
            }
            break;
            case S_ALARM: // Fall through
            default:
            {
                ; // Event unexpected in this state, do nothing
            }
            break;
        }
    }
    else if(hcsr04_distance_cm < threshold_cm)
    {
        // Check the current state and take the appropriate action
        switch(state)
        {
            case S_IDLE:
            {
                // Set the next state
                state = S_CLOSE;
                usart0_transmitStr("state = S_CLOSE\r\n");
            }
            break;
            case S_CLOSE: // Fall through
            case S_ALARM: // Fall through
            default:
            {
                ; // Event unexpected in this state, do nothing
            }
            break;
        }
    }
    else
    {
        // Check the current state and take the appropriate action
        switch(state)
        {
            case S_CLOSE:
            {
                // Set the next state
                state = S_IDLE;
                usart0_transmitStr("state = S_IDLE\r\n");
            }
            break;
            case S_IDLE: // Fall through
            case S_ALARM: // Fall through
            default:
            {
                ; // Event unexpected in this state, do nothing
            }
            break;
        }
    }

    // Update the LCD, no matter the state

    // Create ASCII strings
    char str1[20];
    char str2[20];

    sprintf(str1, "%03d cm to object", hcsr04_distance_cm);
```

```
    sprintf(str2, "%03d cm threshold", threshold_cm);

    // If alarm is active, overwrite the string
    if(state == S_ALARM)
    {
        sprintf(str2, "     ALARM!     ");
    }

    // Show the information on the LCD
    lcd_cursor(0,0);
    lcd_print(str1);

    lcd_cursor(1,0);
    lcd_print(str2);

    _delay_ms(100);
}

int main(void)
{
    setup();

    // Global interrupts enable
    sei();

    while (1)
    {
        loop();
    }
}
```

10.4 Assignment

Use the code examples from the previous paragraph to fulfil the following assignment.

Finish the project from the previous section by implementing the files *hcsr04.h* and *hcsr04.c* and add an additional (fake) sensor that sets the distance threshold.

SMART technical specification		
#	**MoS CoW**	**Description**
T1	**M**	**Use the following hardware components:**
T1.1	M	LCD display
T1.2	M	HC-SR04 ultrasonic range finder
T1.3	M	Potentiometer
T1.4	S	Breadboard

SMART functional specification		
#	**MoS CoW**	**Description**
F1	**M**	**Print a welcome message to the Terminal Window application as soon as the application starts. The welcome message shows the following information:**
F1.1	M	Your name
F1.2	M	Your student number
F1.3	M	The following phrase: “Week 6. Sensors and Actuators”
F2	**M**	**The LCD shows information**
F2.1	M	Line one shows the threshold distance in centimetres.

SMART functional specification		
#	**MoS CoW**	**Description**
F2.2	M	Line two shows the distance in centimetres measured by the ultrasonic range sensor.
F3	**M**	**The alarm is triggered if the measured distance is below a configurable threshold.**
F3.1	M	The threshold is configured with a potentiometer. The potentiometer range is mapped from 0 to 500. This value is the threshold distance in centimetres.
F3.2	M	If the measured distance by the ultrasonic range sensor is below the threshold distance, an alarm is raised.
F3.2.1	M	Line one of the LCD still shows the threshold distance and also the phrase 'ALARM!'
F3.2.2	M	The alarm is NOT deactivated when the distance drops below the threshold.
F3.2.3	M	The LED connected to PB5 is on.
F4	**M**	**The alarm can be deactivated by the user.**
F4.1	M	Pressing the switch SW0 deactivates the alarm.
F4.2	M	The phrase 'ALARM!' is not shown on the display anymore.
F4.3	M	The LED connected to PB5 is off.

11 Week 7. Project Work

During this final week there will be no new subjects. An example exam will be discussed and we will work on the project.

12 References

Datasheet. (2015, 01). *ATmega328P datasheet.* Retrieved June 2021, from https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

HC-SR04. (n.d.). *Ultrasonic Ranging Module HC-SR04.* Retrieved June 2021, from https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf

User Guide. (2017). *ATmega328P Xplained Mini - User Guide.* Retrieved June 2021, from http://ww1.microchip.com/downloads/en/devicedoc/50002659a.pdf