HOCHSCHULE
RHEIN-WAAL
Rhine-Waal University
of Applied Sciences

LABORATORY

Microcontroller

2

EXPERIMENT:

Input and Output

1 Digital I/O

Connect the LED and the button on the board like in Figure 1.

Open the directory "Template/Experiment2" and read the simple C program.

Task 1:
The LED should be on if the button is pressed and off if the button is not pressed. Compile, flash and test the program.

Remember: The buttons on this board are only able to establish a connection to ground. If we simply connect it to the microcontroller, this turns out to be the situation like in Figure 2 and Figure 3.

1.1 Debouncing

Task 2:
Edit the program to toggle the LED with a button push.

You will get a problem, the LED can toggle many times on every key press event. To solve this problem, program a software that works like a debouncing circuit. Debouncing is nessasary for the rising and the falling edge. The LED should only toggle one time per button press, independend how long the button is pressed.

It is not a good idea to program a waiting loop by hand. For example a simple delay loop:

```
uint16_t i;

for (i = 0; i < 100; i++);
```

This solution is depending on the CPU clock speed and the compiler. So it is not possible to wait for a defined time. The GCC compiler uses an optimizer to make the program faster/smaller. This optimizer replaces this code! The optimized version is:

```
uint16_t i;

i = 100;
```

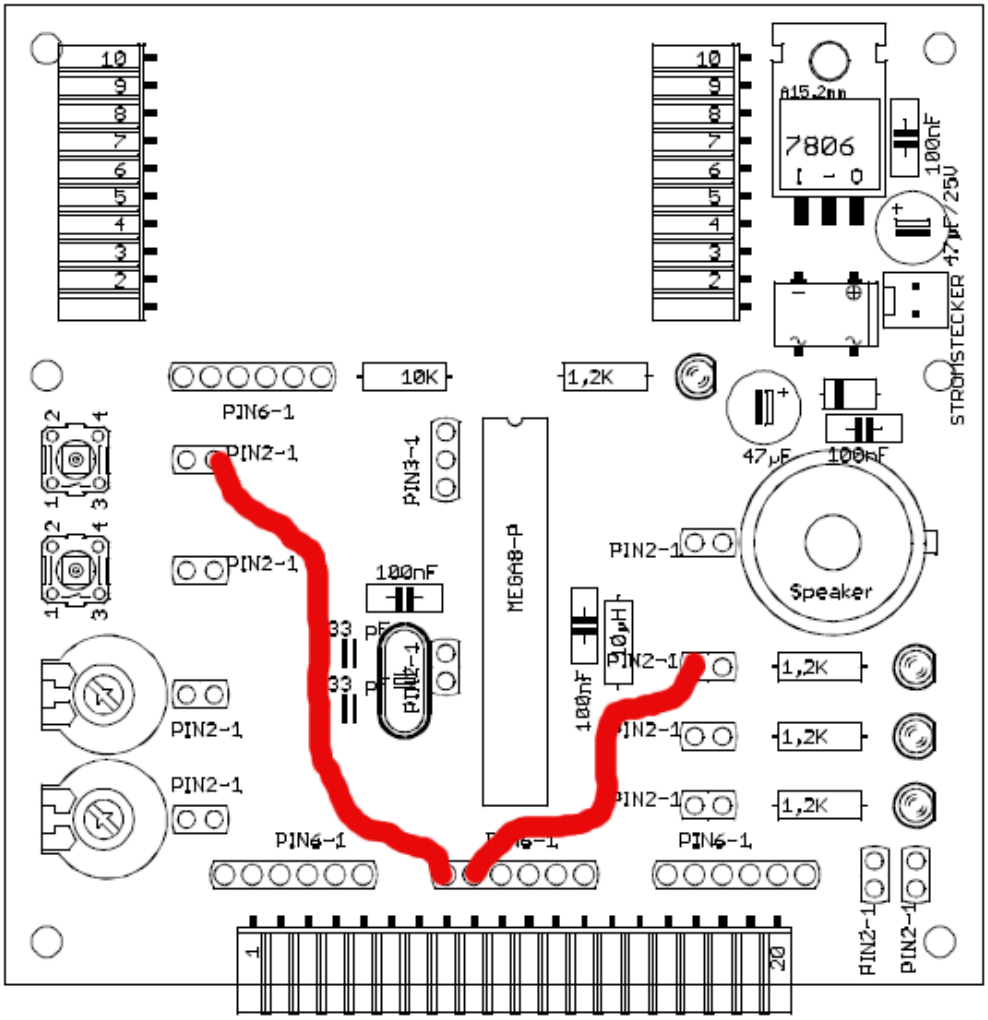
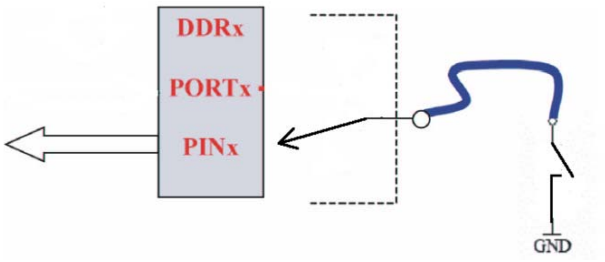
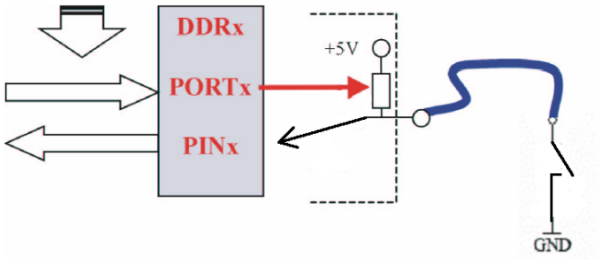


Figure 1: Simple connection with one button and one LED



You will notice, that there is no defined level, once we open the switch. The voltage level is floating. Fortunately the AVR provide a build in solution: Pullup resistors, or “Pullups”, see Figure **??**.

Figure 2: Button, pullup is missing



If we write a “one” to the PORT register, although the corresponding line should be an input, there is a small pullup enabled inside the AVR.

Figure 3: Button, internal pullup is used

2 Analog I/O

Digital signals are only true (**1**) or false (**0**). But in the real world many signals can also be "a little bit" open or closed. To handle analog values like

- lightness
- rotation speed
- temperature
- voltage
- current
- compression

it is not adequate to use only digital input/output. We also have to know about analog input/output.

2.1 Analog output

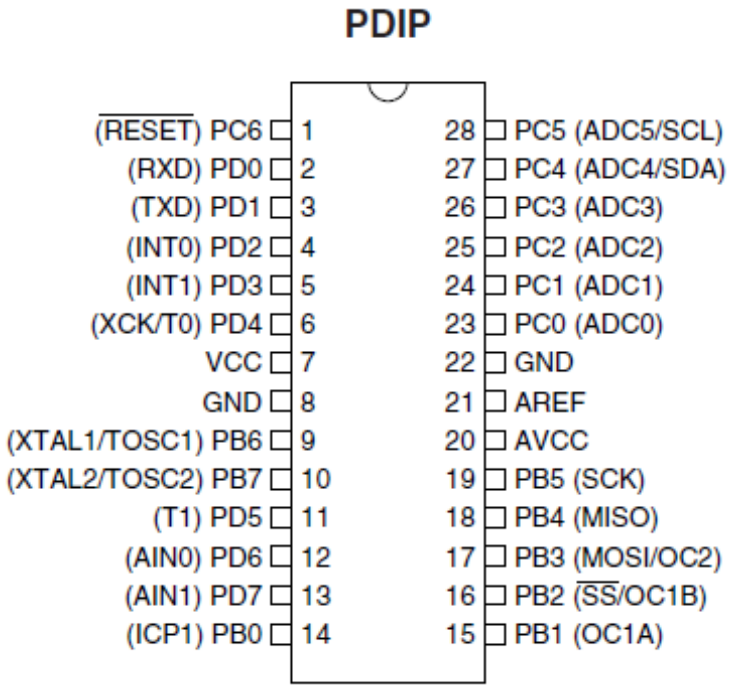
The Atmel ATmega88PA has no real analog output. It emulates this with Pulse-Width modulation (PWM).

Task 3:
To understand PWM write a simple program that enables and disables the LED in an infinite loop when the button is pressed, otherwise the LED should be constantly on.

Compare the brightness when the button is pressed with the brightness when the LED is constantly on.

This simple program uses a method called Software-PWM. Most microcontrollers are able to generate PWM signals with build in hardware. This is more efficient. The Atmel ATmega88PA is able to generate PWM with Timers/Counters. We will become acquainted with this in upcoming labs. With the counters it is possible to vary the on and off time. That is the normal way to set the output level.

A PWM signal generator with this circuit is called DAC (digital to analog converter).



The Atmel ATmega88PA has 6 Channels for analog input (PIN 23 to 28)

Figure 4: Atmel ATmega88PA

2.2 Analog input

Most real world data is analog. Whether it is temperature, pressure, voltage, etc, their variation is always analog in nature. For example, the temperature inside a boiler is around $800\,^{\circ}\mathrm{C}$. During its light-up, the temperature does not reach $800\,^{\circ}\mathrm{C}$ directly. If the ambient temperature is $400\,^{\circ}\mathrm{C}$, it will start increasing gradually to $450\,^{\circ}\mathrm{C}$, $500\,^{\circ}\mathrm{C}$ and finally reaches $800\,^{\circ}\mathrm{C}$ after a period of time. This is an process with analog behaviour and so the data is also analog.

Now, we have to process the data, which we have received. But pure analog signal processing is quite inefficient in terms of accuracy, speed and desired output (there are analog "computers", which consist of a lot of op amps and execute calculations in a purely analog way). Hence, we convert them to digital form using an **A**nalog to **D**igital **C**onverter (ADC).

The ATmega88PA has 6 ADC input channels, see Figure 4.

There are three basic steps to get a real world value into the microcontroller, this process is called **Signal Acquisition Process**, see Figure 5:

- In the real world, a sensor senses any physical parameter and converts into an equivalent analog electrical signal.
- For efficient signal processing, this analog signal is converted into a digital signal using an ADC.
- This digital signal is then fed to the Microcontroller (MCU) and is processed accordingly.

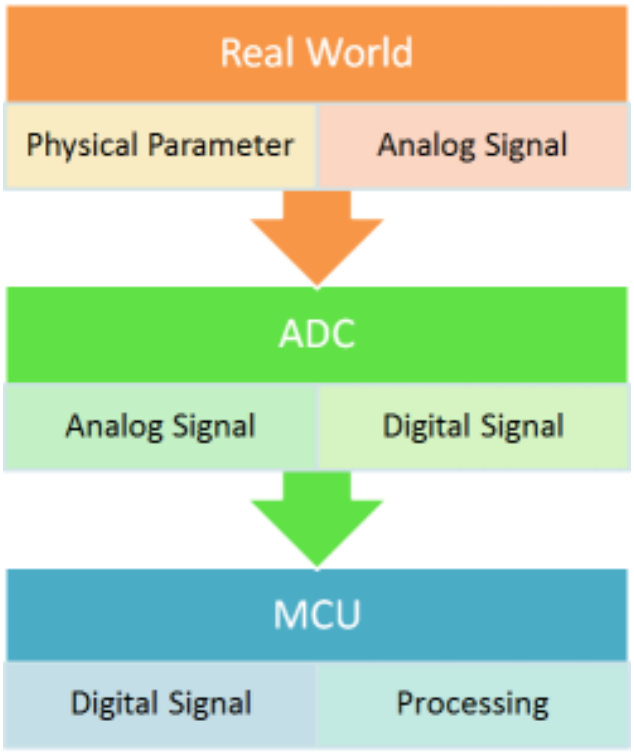


Figure 5: Signal Acquisition Process

2.2.1 Signal Acquisition Process

In general, sensors provide an analog output, but a MCU is a digital one. Hence we need to use an ADC. Fortunately, there is an inbuilt ADC in the ATmega88PA (and in mostly all other AVRs). PORTC contains the ADC pins.

Some of the features of the ADC are as follows (page 237 from datasheet):

- 10-bit Resolution
- 0.5 LSB Integral Non-linearity
- ± 2 LSB Absolute Accuracy
- $13\mu s$ - $260\mu s$ Conversion Time
- Up to 15 kSPS at Maximum Resolution
- 6 Multiplexed Single Ended Input Channels
- 2 Additional Multiplexed Single Ended Input Channels (TQFP and QFN/MLF Package only)
- Optional Left Adjustment for ADC Result Readout
- 0 - VCC ADC Input Voltage Range
- Selectable 1.1V ADC Reference Voltage
- Free Running or Single Conversion Mode

- Interrupt on ADC Conversion Complete
- Sleep Mode Noise Canceler

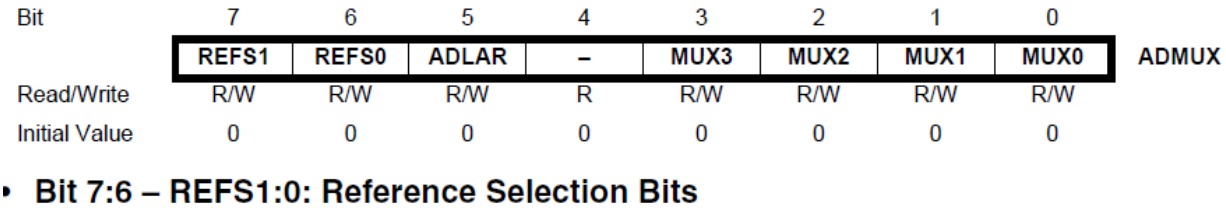
Suppose we use a 5V reference. In this case, any analog value in between 0 and 5V is converted into its equivalent ADC value as shown in Figure 5. The 0-5V range is divided into $2^{10} = 1024$ steps. Thus, a 0V input will give an ADC output of 0, 5V input will give an ADC output of 1023, whereas a 2.5V input will give an ADC output of around 512. This is the basic concept of ADC.

By the way: the type of ADC implemented inside the AVR MCU is of Successive Approximation type.

Apart from this, the other things that we need to know about the AVR ADC are:

- ADC Prescaler
- ADC Registers – ADMUX, ADCSRA, ADCH and ADCL

2.2.2 ADC Registers



Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7:6 – REFS1:0: Reference Selection Bits**

Figure 6: ADMUX Register

The ADMUX Register (see Figure 6) is one of the control registers for the ADC. It holds the MUX bits, which select the channel (0 to 5 in our case), the reference selection bits, which specify which reference is to be used for the ADC and the ADLAR (ADC Left Adjust Result) bit. The last one specifies how the 10 bit result is adjusted in the two result registers ADCL and ADCH:

Let's assume the result would be 578 (1001000010 binary). With ADLAR set to one, the ADCH and ADCL would look like this:

ADCH: 1 0 0 1 0 0 0 0
ADCL: 1 0 - - - - - -

With ADLAR set to zero:

ADCH: - - - - - - 1 0
ADCL: 0 1 0 0 0 0 1 0

Table 24-3. Voltage Reference Selections for ADC

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal V_{ref} turned off
0	1	AV_{CC} with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 1.1V Voltage Reference with external capacitor at AREF pin

Figure 7: Voltage Reference

Table 75. Input Channel Selections

MUX3..0	Single Ended Input
0000	ADC0
0001	ADC1
0010	ADC2
0011	ADC3
0100	ADC4
0101	ADC5

Figure 8: MUX settings

In Figure 7 is, how to select the **REFS1** and **REFS0** bits. In the lab, we are going to use the AVCC as reference only, because the potentiometers on the board are connected between 5V and ground, in order to provide 0 to 5 V on their wiper terminal.

The table in Figure 8 shows, how to set the MUX3 to MUX0 bits in the ADMUX register, corresponding to the desired input channel.

If you connect the analog input signal e.g. to PC3 (where the ADC3 is located), you would have to select ADC3 using the MUX bits. In this case, you would set MUX1 and MUX0 to 1. This can be done in the well known way:

```
ADMUX |= (1 « MUX0)|(1 « MUX1);
```

To enable the value measurement it is necessary to set the ADC Control and Status Register (ADCSRA), see Figure 9:

- **ADEN: ADC Enable** Writing this bit to one enables the ADC. By writing it to zero, the ADC is turned off.
- **ADSC: ADC Start Conversion** In Single Conversion mode, write this bit to one to start each conversion. In Free Running mode, write this bit to one to start the first conversion. The first conversion after ADSC has been written after the ADC has been enabled, or if ADSC is written at the same time as the ADC is enabled, will take 25 ADC clock cycles instead of the normal 13. This first conversion performs initialization of the ADC. ADSC will read as one as long as a conversion is in progress. When the conversion is complete, it returns to zero. Writing zero to this bit has no effect.
- **ADATE: ADC Automatic Update** When this bit is set (one) the ADC operates in Free Running mode. In this mode, the ADC samples and updates the Data Registers continuously. Clearing this bit (zero)

Bit	7	6	5	4	3	2	1	0	
(0x7A)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
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	

Figure 9: ADC Control and Status Register

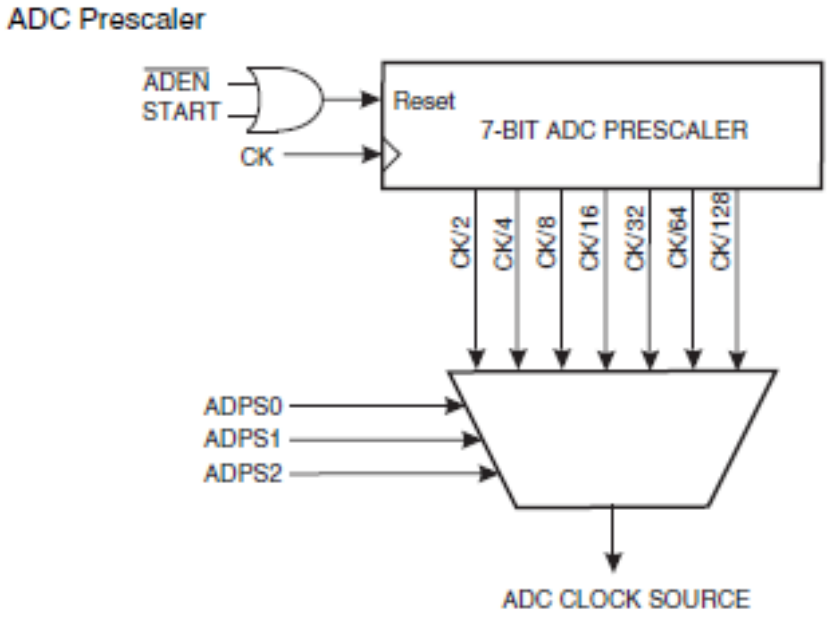


Figure 10: ADC Prescaler

will terminate Free Running mode. There are more automatic update modes, see the datasheet on page 251.

In order to keep it simple in this lab, we are going to use the Free Running Mode

The ADC of the AVR converts analog signal into digital signal at some regular interval. This interval is determined by the clock frequency. In general, the ADC operates within a frequency range of 50kHz to 200kHz. But the CPU clock frequency is much higher (3.6848 MHz). To achieve it, frequency division must take place. The ADC-prescaler acts as this division factor. It produces desired frequency from the external higher frequency. There are some predefined division factors – 2, 4, 8, 16, 32, 64, and 128. For example, a prescaler of 64 implies F_ADC = F_CPU/128. For F_CPU = 16MHz, F_ADC = 16M/128 = 125kHz.

Now, the major question is... which frequency to select? Out of the 50kHz-200kHz range of frequencies, which one do we need? Well, the answer lies in your need. There is a trade-off between frequency and accuracy. The greater the frequency, the poorer the accuracy and vice-versa. Therefore, if your application is not sophisticated and doesn't require much accuracy, you could go for higher frequencies.

To set the prescaler at the Atmel ATmega88PA microcontroller, set the bits **ADPS0**, **ADPS1**, **ADPS2** in the ADCSRA register, see Figure 10 and Figure 11.

Inside the hardware from the Atmel ATmega88PA is a build-in securtiy lock. This lock prohibits to read

Table 24-4. ADC prescaler selections.

ADPS2	ADPS1	ADPS0	Division factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Figure 11: ADC Prescaler Bits

an inconsistent value between reading ADCH and ADCL. If a read operation on ADCL is detected **both** registers will be locked until there is a read operation on ADCH!

So this code will **not** work:

```
uint16_t value;
value = ADCH * 256;
value += ADCL;
```

To solve this read this two registers in the correct order or better use ADCW:

```
uint16_t value;
value = ADCW;
```

2.2.3 ADC tasks

Before you change your last program, always make a copy. At the end of the experiment you should be able to show and explain all programs of an experiment.

A template for this task is the directory in "Template/Experiment2".

Connect 3 LEDs to PORTB, B1, B2 and B3. Connect one potentiometer to ADC3.

Task 4:
Write a program, which

- initialises the ADC, use the free running mode
- reads the voltage on the potentiometers wiper
- outputs the voltage range on the 3 LEDs
  - green: 0 – 1.66V
  - yellow: 1.66 – 3.32V
  - red: 3.32 – 5V

Task 5:
In order to use a “real” sensor: connect the photo sensor to ADC3. The photo sensor delivers:

- in the dark: appr. 2.55 V
- ambient light: appr. 4.20 V

Task 6 (Optional):
There are two files available in the experiment2 directory: lcd.c and the corresponding header file lcd.h. They are already customized for the myAVR LCD 2.5 module. You can add them to your project, if you include "lcd.h". Check if the makefile has to be modified. Have a look at the header file and find out, how to output a string on the display.
Write a program that shows the ADC value on the display.