

# LABORATORY

Microcontroller

1

EXPERIMENT:

## Basic Microcontroller programming

Write your name in every sourcefile you edit and compile the code with your matriculation number (variable in the template). All files should contain all names of the persons who made changes. Do use the @author tag for this (as available in the headline of template files).

# 1 Micro...

A few classes of controllers and systems are available on the market:

- **Microprocessor:** A ‘common’ processor, eg. the CPU in a PC. The communication to peripherals is done by additional parts using a bus.
- **Microcontroller:** Is a microprocessor, which already contains all necessary components to make it a ready to use “one-chip-micro-computer” (also called “system-on-chip”). It contains a microprocessor, some memories, interfaces, timers and an interrupt-controller. It is able to perform measurements using digital and analog inputs. Output hardware helps to control external equipment.
- **Signal processor, Digital Signal processor (DSP), Mixed-Signal-Controller:** DSPs are microcontrollers which can compute digital and analog signals very fast.
- **Embedded Processor, Embedded System:** A good example is a smartphone. Very often there is an ARM-controller working in these systems to control all the functions of the phone (like the display, touch screen, music player, camera and the wireless communication). The controller itself is part of the device it controls.

In this lab we start with microcontrollers. In upcoming labs, you will become familiar with DSPs and embedded systems. The microcontroller (AtMega88PA) we are going to use is based on the Harvard architecture.

There are four sub-components in this architecture: 1) Memory (for program code, and for variables inside the program), 2) Input/Output (called “IO”), 3) Arithmetic-Logic Unit (ALU), 4) Control Unit.

Often omitted: There is a 5th key player in this operation: a bus, or couple of wires, that connects the components together and over which data flows from one sub-component to another. Internally used signals are all binary-coded. The processor uses words with a defined length (bus system, registers, memory).

The program instructions are processed in a sequential order:

1. Power-on or reset,
2. load instruction-pointer with fixed start address (0x00),
3. read instruction (from instruction-pointer),
4. decode instruction,
5. perform instruction,
6. increment instruction-pointer (+1), continue with 3.

It is still a little bit like in Figure 1 and Figure 2:

The processor is clock-controlled, the sequential processing can be modified by “jump instructions”: Instruction pointer = jump address. This jump can be performed either independent of or dependent on a certain condition.

We can distinguish between several processors based on the maximum number of bits they can process in one step: (4 bits), 8 bits, 16 bits, 32 bits, 64 bits and so on. You might be used to 32 bit and 64 bit systems

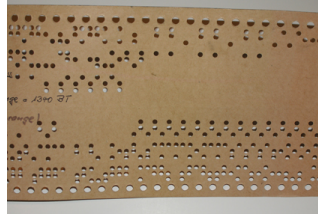


Figure 1: Punch card.



Figure 2: Using punch cards.

(eg. personal computers). But what about these bits? An 8 bit microcontroller can process 8 bits at one stroke. The biggest unsigned number which can be expressed by 8 bits, is 255.

$$max = 2^n - 1$$

This means: The 8 bit controller for example can add two numbers (which are both smaller than 255) with a single instruction, as long as the result is also smaller than 255. If the numbers are greater, more instructions and more clock cycles are required. A processor usually needs an external clock source. Because the internal units of the processor are neither working at an infinite speed, nor at an equal speed, this clock synchronizes them. It's like when assigning jobs to several groups of students and expecting the results back on next Monday, so as to put together in a single project. All these groups of students might be working at different speeds, however, the project is concluded on Tuesday – synchronized.

The maximum frequency of processors ranges between 1 MHz to more than 4 GHz (4 000 000 000 operations per second!). Modern microcontrollers work in frequency ranges of 1 to 300 MHz. Be careful: A microcontroller running at 20 MHz does not necessarily mean, it is faster than another one running at 10 MHz. It depends on how many instructions per second (MIPS – Mega Instructions per Second) it can perform. CISC processors (Complex Instruction Set Computer) often need more clock cycles to execute a single instruction. The available instructions of the CISC are more complex than those of the RISC (Reduced Instruction Set Computer), which uses simpler and lesser instructions.

The result is that the processor often needs only one cycle to execute a (simple) single instruction. Hence, more instructions are needed to do the same job. For this lab we are going to use AVR (Advanced RISC) controllers. The AVR core was developed by two students at Trondheim university in Norway.

## 1.1 For which purposes can we use microcontrollers?

Just three examples of devices with microcontrollers in them:

- Traffic lights,
- TV remote,
- MP3 player.

## 2 Peripherals of an AVR microcontroller

Peripherals are parts of microcontrollers. Without them a microcontroller would be only a microprocessor. The AVR microcontrollers have the following peripherals:

- **Digital Inputs and Outputs (Digital I/O):** The most important I/Os are the digital ones. They are called PORTS and are able to represent digital signals (Hi and Low; 1 and 0; 5 and 0 Volts) and read digital signals. A port consists of a number of lines (Pins), usually 4, 6, or 8. Being able to deal with 0 and 5 volts (only), these ports are used to both produce as well as to read on- and off -signals, pulses and frequencies. However, they are not able to read analog voltages in-between. For measuring e.g. 2,592 volts, or to compare a voltage to 3,214 Volts (bigger / smaller) an analog to digital converter or an analog comparator has to be used.
- **Analog to digital converter (A/D converter):** To read an analog voltage level (not only high or low), the A/D converter can be used. The AVR ATmega controllers carry a 10 bit A/D converter with them. Voltage levels are represented as 10 bits. Due to using an 8-bit controller, two 8-bit numbers will be used to store the A/D result. That means, 6 bits are occupied, but not used. This is called “overhead”.
- **D/A converter:** The converse of the A/D converter, converts a digital signal into an analog voltage. There are lots of possible ways to do so. Standalone AVRs only provide a solution of applying a PWM signal on an R-C network. This method uses only one pin of the controller.
- **Timer:** Often a microcontroller is supposed to do things with a special timing (e.g. waiting for exactly 3.2 ms, until another action takes place, or pulling a pin “high” for a defined period of time). Without exact timers this would not be possible. Do not use a for-loop, which counts to 10000, like it is sometimes done when programming in C on a personal computer. There are special components designed for exactly this task build-in in the microcontroller.
- **Send and receive modules (RX/TX):** To communicate with either the user, or another system there are interfaces provided in AVRs. We are going to use the most common one for user communication, namely the UART (Universal, Asynchronous, Receiver, Transmitter). The UART can be used as RS232 interface and can be connected to a PC via a converter (e.g. a MAX232, or a CP2102 USB bridge). There are AVRs available with a build-in USB interface. But if someone simply wants to understand the basics of serial communication, the RS232 is more suitable, because there is less protocol-overhead, which makes it possible to “observe” the communication manually (e.g. with an oscilloscope).
- **RAM:** Data can be written to and read from the Random Access Memory. The RAM is volatile and its content will be lost or erased if its supply voltage is switched off. The RAM is very fast.
- **Registers:** Registers are memory cells, which are directly connected to the CPU core. They hold operands and results at the point in time in which the CPU processes data. Registers are designated with characters or just numbered (R1, R2, .. , R32).
- **ROM:** Read Only Memory. As its name implies, this is a memory where information is stored once. From that point on, the microcontroller can only read the information. No manipulation can be done any more.
- **EEPROM:** Electrically Erasable Programmable Read Only Memory. This is similar to the RAM, but non volatile. The content is not erased if the power is switched off. The number of writing actions are limited (around 100000 cycles). The controller can itself store information in here.

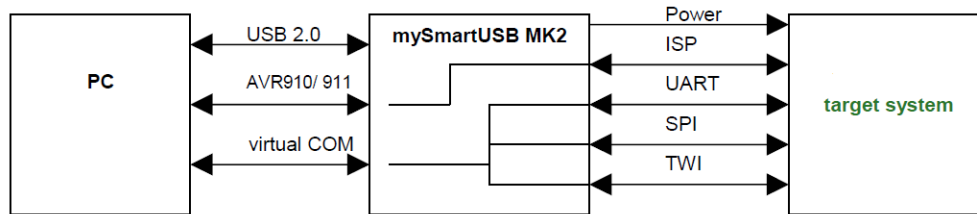


Figure 3: My Smart USB MK2 (ISP programmer).

- **Flash:** A sort of EEPROM, but much faster. The number of writing actions here is limited and ranges from 1000 to 10000 cycles. Here, in the flash memory the actual “program” is stored. When we transfer the program from the PC to the controller, a special application writes it in here.

## 2.1 Program Transfer

But how do we transfer a program into a microcontroller? Obviously, there is no CD-drive or hard disk. The solution is the programmer, often called ISP (In System Programmer). Luckily, we have USB programmers nowadays. In earlier times, we had to build a programmer for the serial port (RS232) or a parallel port. You can see an example in Figure 3.

This programmer is multifunctional. It provides a power supply and a serial interface to the controller as well (e.g. USB-serial bridge, or virtual COM). This means we are not only able to transfer a program to the microcontroller, but can also “talk” to it using the same USB connection on the computer! See Figure 3.

## 3 Program Code

What does a program look like? Here is an example of a very simple program, which toggles an LED connected to the pin B1, if a button connected to pin D2 is pressed:

```

:1000000012C024C023C022C021C020C01FC01EC0F7
:100010001DC01CC01BC01AC019C018C017C016C014
:1000200015C014C013C011241FBECFE5D4E0DEBF3D
:10003000CDBF10E0A0E6B0E0E8E9F0E002C0059036
:100040000D92A236B107D9F702D024C0D9CF1CD067
:100050008091600042E028EC30E001C080E08299AD
:100060000DC0882359F088B3842788BB80E293E0D1
:10007000F9013197F1F70197D9F7F0CF829BEFCFD4
:10008000882369F781E0EBCF8A988B98929A939AAC
:08009000B99A0895F894FFCF1E
:02009800010065
:00000001FF
  
```

[first\\_c.hex](#)

The file format is Intel hex. It contains the machine-readable instructions and initialization values.

As there is (mostly) no way to develop the software on the microcontroller (How should that be possible: There is no keyboard, no mouse, no CRT monitor and no operating system on the microcontroller system), the typical way is the following:

We need to compile (translate) the source code into machine-readable code on a (powerful) system, that itself cannot understand/read this machine code! This is called a cross-compiler.

### 3.1 GNU Toolchain

The compiler we are going to use is the GCC (Gnu Compiler Collection). GCC can compile source code for various targets including x86, ARM, Blackfin and Atmel AVR. Equipped with libraries for AVR, it becomes a quite comfortable cross-compiler. The compiler is part of the so called “Toolchain”.

The GNU - Toolchain:

- **GNU make:** Automated tool for compilation,
- **GNU CC:** The compiler,
- **GNU binutils:** assembler, linker,
- **a text editor:** here we will write our source code,
- **libraries:** pre-written code, subroutines, values, definitions customized for a target system,
- **“burn”-software:** write (often called “flash”) the compiled program into the microcontroller.

We are going to take a look at the “flash”-software. An Avrdude is included in the winAVR / GNU toolchain. It is a command-line tool that might be scary, if you see it for the first time, but this is the best way to learn what this tool does.

Right now, there is no program running on the microcontroller (or it actually may be some code from the last year). We are going to put a pre-compiled software into it. Connect one button to PORTD / PIN 2 (in short: D2), and one LED to PORTB / PIN 1 (in short: B1). If you power up the board, nothing is going to happen. Now we will use **avrdude** to flash the first simple test program "program.hex" to the microcontroller. The first step is to open a terminal. Inside this terminal, you can enter commands.

#### Task 1:

To open a terminal, use the shortcut STRG + ALT + T. All files you need are inside a (sub)directory. For all experiments there are pre-written files (templates) available. To go into the directory with all templates enter the command: `cd Templates`. To change into the directory for this experiment enter `cd Experiment1`. This directory already contains the file `program.hex`.

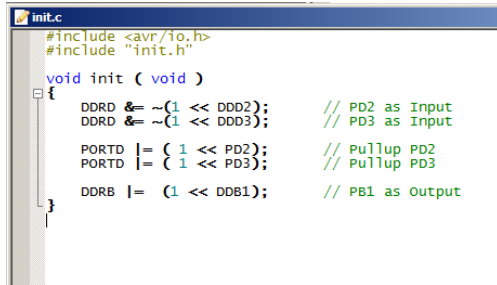
Write (and create the screenshot of this step for the lab report):

```
avrdude -p m88p -c avr911 -P /dev/ttyUSB_MySmartUSB -U flash:w:program.hex:i
```

to transfer the file into the microcontroller.

**Hint:** In the future, you may also comfortably navigate the file explorer to the folder of interest and select the context menu (right mouse click) the option “Open in terminal”.

Have a look at `avrdude -help` (write this line into the terminal window)



```

#include <avr/io.h>
#include "init.h"

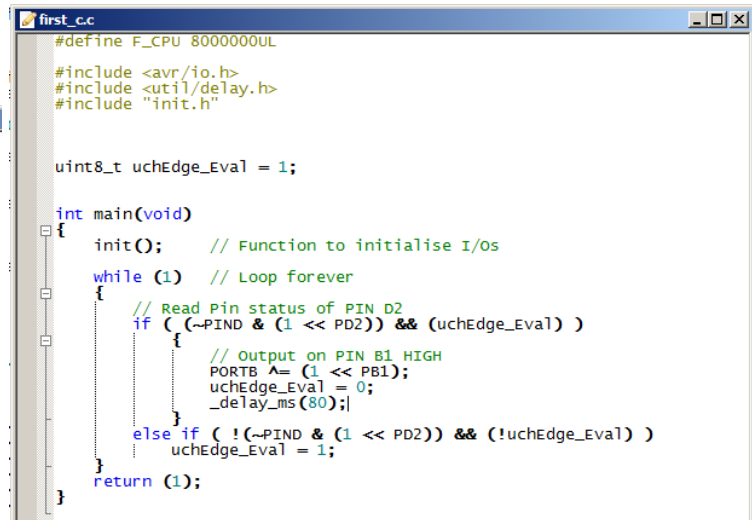
void init ( void )
{
    DDRD &= ~(1 << DD02);    // PD2 as Input
    DDRD &= ~(1 << DD03);    // PD3 as Input

    PORTD |= ( 1 << PD2);    // Pullup PD2
    PORTD |= ( 1 << PD3);    // Pullup PD3

    DDRB |= (1 << DD01);    // PB1 as output
}

```

Figure 4: File init.c.



```

#define F_CPU 8000000UL

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

uint8_t uchEdge_Eval = 1;

int main(void)
{
    init();    // Function to initialise I/Os

    while (1) // Loop forever
    {
        // Read Pin status of PIN D2
        if ( (~PIND & (1 << PD2)) && (uchEdge_Eval) )
        {
            // Output on PIN B1 HIGH
            PORTB |= (1 << PB1);
            uchEdge_Eval = 0;
            _delay_ms(80);
        }
        else if ( !(~PIND & (1 << PD2)) && (!uchEdge_Eval) )
        {
            uchEdge_Eval = 1;
        }
    }

    return (1);
}

```

Figure 5: File first\_c.c code.

- **-U flash** does some action to the flashmemory,
- **:w** is an action avrdude shall perform,
- **"program.hex"** is the hex-file we want to write,
- **:i** tells avrdude, that our file is in Intel hex format.

After the program has been successfully flashed, it is possible to toggle the LED with the button.

Now you know the basics, how to transfer a compiled program to a microcontroller (MCU). But the compiled code is not human-readable. So what do you do? (if you want to use for example, two buttons). A modification of the hex file is almost impossible, unless you know exactly what is going on there and are able to read machine-code. The solution is another level of programming language. A "higher" level, which is readable for us humans. Here our compiler is the key-player. It's the link between human-readable and machine-readable language. Let's have a look at the C program, which was the source for first\_c.hex / program.hex (just flashed to the MCU), see Figure 4 and Figure 5.

You can easily find some operations, which deal with PORT-Registers (PORTB) and PIN-Registers (PIND) in the main routine. Our LED is connected to PORTB. The button is connected to PORTD.

What you see in these few lines is a query or detection on the PIN-Register and some dependent action on the PORTD-Register.

Do not bother much about the code for now, but you should rather understand the bitwise logic operations, like:

&, ^, ~

and the shift operations (1 << XX). We will definitely need this knowledge, without which, microcontroller programming is not possible.

## 4 Example Programs

The downside of the MYAVR board is that it has no isolation (insulation). Never place this board on conductive things (like a pen or metal **e.g. of your notebook**). The table in the lab is not conductive.

Now let us check several simple programs. Before flashing and running the programs, connect the microcontroller as follows:

- Pin B4 with key 1,
- Pin B5 with key 2,
- Pin B0 with the red LED,
- Pin B1 with the yellow LED,
- Pin B2 with the green LED,
- Pin C0 with the potentiometer poti 1,
- Pin C3 with the sounder (on some boards labelled with “Summer”).

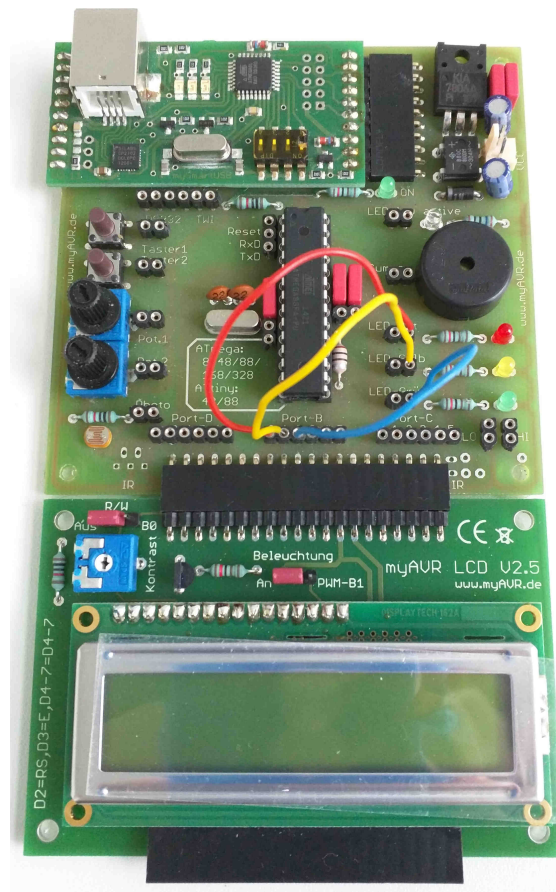


Figure 6: Example for several connections (just B0, B1, B2) on the myAVR board.



### Task 2:

After these connections are done, flash the example programs and test the result. **Describe every program with one-two sentences in the lab report.** Please select **two** most impressive programs (in your opinion).

The commands to flash a program are:

- *make flash1*  
This program enables or disables the LEDs controlled by the keys (key one: all LEDs on, key two: all LEDs off).
- *make flash2*  
This program simulates a simple traffic light. Keys have no function.
- *make flash3*  
Two different tones will be generated by holding the keys one and two.
- *make flash4*  
Press key one to play a little bit of music.
- *make flash5*  
Moving LCD Display Output. Press key one or key two to re-start.
- *make flash6*  
This program generates a square wave signal with a duty cycle of 50%. You can set the output frequency with the potentiometer (Pot. 1), keys have no function.

## 5 C Programming

We are going to start with a simple program. Change into the subdirectory 'Task3' with the *cd* command. Open the file "main.c" with the text editor of your choice. Write your name and your matriculation number in the corresponding areas of the template.

Have a look at the line `if ((~PIND & (1 << PD2)) && (~PIND & (1 << PD3)))`

There you can find operations like: "~", "&", "&&" and "<<".

- "~" (called tilde) is a bitwise negation,
- "&" is a **bitwise and** operation (bit by bit),
- "&&" is a **logical and** operation (true or false),
- "<<" is a **shift** (to the left in this case) operation.

Bytes can represent a logical value. Only a byte that is zero is **false**. All other values are treated as **true**. Logical operations have only logical results (true/false).

**Hint:** The buttons on this board are only able to establish a connection to ground (as they are soldered accordingly). If we simply connect it to the microcontroller, to be like the situation in Figure 7 (there is

no defined level, once we open the switch as the voltage level is floating. Fortunately, the AVR provides a build-in solution: Pullup resistors, or “Pullups”, see Figure 8 - if we write a “one” to the PORT register, a small pullup is enabled inside the AVR, although the corresponding line should be an input).

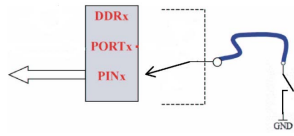


Figure 7: Button, pullup is missing.

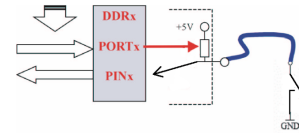


Figure 8: Button, internal pullup is activated and used.

## 5.1 Compile

Back to our main.c. First we are going to call the compiler and the linker manually on the command line:

```
avr-gcc -g -Os -mmcu=atmega88pa -c main.c
```

- **avr-gcc** GNU compiler call,
- **-g** add debugging symbols in the binary file,
- **-Os** optimisation level (for small size),
- **-mmcu=atmega88pa** target system (atmega88pa),
- **-c** compile only,
- **main.c** file to compile.

### Task 3:

Open the command prompt (STRG + ALT + T), navigate to the folder that contains the .c file (Template/Experiment1/Task3). Type “ls” to see all files in that directory and use “cd” command to change the directory. Compile the .c file using the previously introduces command. Type “ls” again. Which file(s) have been created now? **Describe it in your lab report.** Provide the screenshots (optional). (To have a closer look at the compiler options *avr-gcc -help* can be called.)

## 5.2 Linking

To link the compiled program use the command:

```
avr-gcc -g -mmcu=atmega88pa -o main.elf main.o
```

- **avr-gcc** GNU compiler call,

- **-g** add debugging symbols in the binary file,
- **-mmcu=atmega88pa** target system (atmega88pa),
- **-o** place the output in,
- **main.elf** filename used for the output,
- **main.o** filename used for the input.

#### Task 4:

Open the command prompt / terminal. Link the .c file (compiled in the previous task) using the command above. Which file(s) have been created now? **Describe it in your lab report.** Provide the screenshots (optional).

Many programmers and programming software only accept Intel hex file format. Therefore we need to “translate” the ELF file into hex using avr-objcopy:

```
avr-objcopy -j .text -j .data -O ihex main.elf main.hex
```

- **avr-objcopy** GNU compiler call,
- **-j** Only copy section <name> into the output,
- **.text** <name> for -j,
- **-j** Only copy section <name> into the output,
- **.data** <name> for -j,
- **-O** Create an output file in format <bfdname>>,
- **ihex** <bfdname> for -O,
- **main.elf** input file,
- **main.hex** output file.

We specify to make a copy of the .text part, which contains the actual program and the .data part, which contains the data. We do NOT copy anything to the controller! Everything still happens only on the PC. We actually only changed the file format of this “executable”. As soon as you finish executing this command, there should be the HEX file in the folder.

#### Task 5:

Open this file in editor or your choice. Provide the screenshot (or the code listing of this hex file) in your lab report.

This method is obviously very time-consuming since you have to compile the source code more than once (and that is always the case!), or the source code consists of few hundred files. Lazy people thought there is a

possibility to automate the compiling and linking. And there is one - it is the MAKE utility. There are certain advantages one has to know when developing software. To make it even more obvious, that this utility is pretty useful, we are going to split our code in two C-files first.

main.c will contain the main, init.c will contain the I/O initialization:

- main.c
  - including header files,
  - main() function and loop with while(1), so-called infinite loop.
- init.c
  - init(void) function, where I/Os are configured in the desired way.
- init.h
  - header file for init.c. This file should contain the prototype of the function init(void),
  - "inclusion lock", or "include guard".

#### **Task 6:**

Edit the main.c with the editor of your choice (vi, nano, gedit). Remove all lines between "// Init START" and "// Init END" and replace it with the function call, just "init();". Now execute this program on the microcontroller. Describe what happens in your lab report.

To execute the program on the microcontroller you need to do these steps:

1. *avr-gcc -g -Os -mmcu=atmega88pa -c main.c*
2. *avr-gcc -g -Os -mmcu=atmega88pa -c init.c*
3. *avr-gcc -mmcu=atmega88pa -o main.elf main.o init.o*
4. *avr-objcopy -O ihex main.elf main.hex*
5. *avrdude -P /dev/ttyUSB\_MySmartUSB -p m88p -c avr911 -Uflash:w:main.hex:i*

## **6 Makefile**

In order to compile the project, it would be necessary to execute the above mentioned commands for the two files. And every time you change the code, you would have to enter these commands once more. That's a bit too much typing, don't you think? Here is the easy way, the makefile.

A Make rule is composed of:

target: prerequisites

<tab> commands

Open the file named makefile (this file will be used by the make utility and it has NO file extension). Put the following lines into that file. The text shifting has to be made with tabs only. Do not use spaces, it will not

work (during manual calls of the gcc components, you noticed the very high number of different flags and settings separated by spaces as there is no clear way to process them. Therefore, only tabulators are allowed here)! This PDF file is generated with L<sup>A</sup>T<sub>E</sub>X so be careful with the copy and paste operations.

```
01 all: main.hex
02
03 main.hex: main.elf
04     avr-objcopy -O ihex main.elf main.hex
05
06 main.elf: main.o init.o
07     avr-gcc -mmcu=atmega88pa -o main.elf main.o init.o
08
09 main.o: main.c
10     avr-gcc -mmcu=atmega88pa -c -Os main.c
11
12 init.o: init.c init.h
13     avr-gcc -mmcu=atmega88pa -c -Os init.c
14
15 program: main.hex
16     avrdude -P /dev/ttyUSB_MySmartUSB -p m88p -c avr911 -Uflash:w:main.hex:i
17
18 clean:
19     rm -f main.hex
20     rm -f *.o *.elf *~
```

### Task 7:

Complete the following tasks (in the Task7 subfolder):

1. Write the makefile, run the automated compilation by typing "make all" on the command prompt (navigate to the working directory first).
2. Flash the hex file (flash means: program the controller with the generated HEX-file.).
3. Invert the behaviour of the **red** LED. This means: previously, in the example program the red LED used the AND or OR function. Change this now to the **NOR function**. Then compile the program again and flash it onto the controller.
4. If that works fine, add a second, **yellow** LED, which indicates that **button one and button two** are pressed (**AND function**).
5. Recompile and flash.
6. Use the third, **green** LED, this LED should represent the **XOR function** for the two buttons. In C, there is no logical XOR operator. To get a logical "A xor B" you can use the following logical operation: "!A != !B".
7. Recompile for last time and flash the hex file to the MCU.

## 7 Modular Programming

### Task 8 (Optional):

Write two more files:

- led.h
- led.c

These files should contain functions to control the three LEDs:

- void ledRed(uint8\_t value)
- void ledYellow(uint8\_t value)
- void ledGreen(uint8\_t value)

Edit the makefile and the main.c to use these functions in your program. The datatype **uint8\_t** is used here as a **bool** (logical value). Zero is false (LED off) and any other value (usually 1) is true (LED on).

## 8 Digital I/O

### Task 9:

The LED connected to pin B1, should be on if the button connected to pin B0 is pressed and off if the button is not pressed. Compile, flash and test the program.

Do not forget to insert your matriculation number in the code (replace the 00000).

### 8.1 Debouncing

You might have get a problem whereby the LED toggles several times on every key press event. To solve this problem, program a software that works like a debouncing circuit. Debouncing is necessary for the rising and the falling edges. The LED should only toggle one time per button press, independent of the duration, for how long the button is pressed.

### Task 10:

Edit the last program to toggle the LED with a button push (toggle operates just like a light switch: once pressed, “on”, by the next button press, it is again “off”).

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

```
uint16_t i;  
  
for (i = 0; i < 100; i++);
```

as this solution is depending on the CPU clock speed and also on the compiler used. The GCC compiler uses an optimizer to make the program faster/smaller. This optimizer replaces this code! The optimized version is as simple as:

```
uint16_t i;  
  
i = 100;
```

## 9 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” opened or closed.

### 9.1 Analog Output

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

#### **Task 11:**

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 as it allows you to use your own code “on top of it”. 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).

### 9.2 Analog Input

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

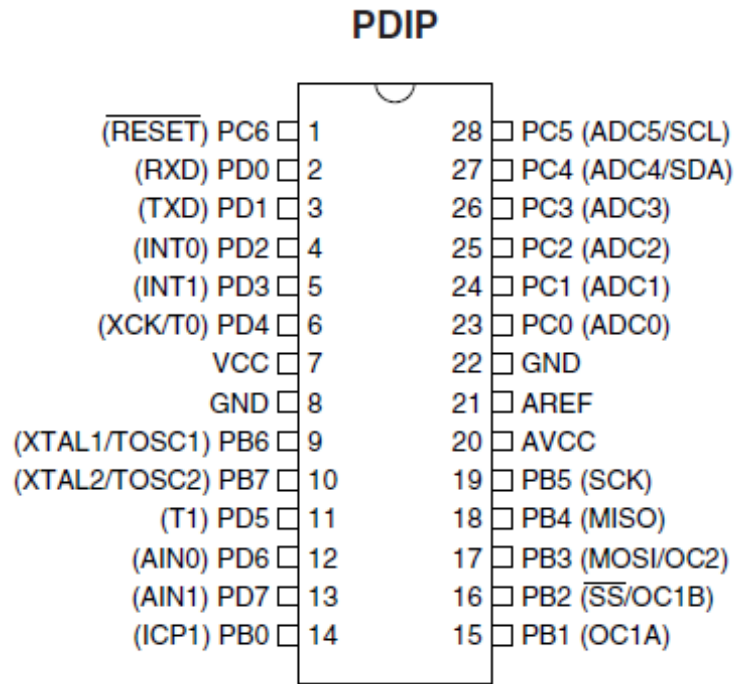


Figure 9: Atmel ATmega88PA has 6 Channels for analog inputs (pins 23 to 28).

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, but you will find them mostly in museums). Hence, we convert them to digital forms using an **Analog to Digital Converter (ADC)**.

The ATmega88PA has 6 ADC input channels, see Figure 9.

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

- 10-bit Resolution,
- 0.5 LSB Integral Non-linearity,
- $\pm 2$  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 5V as a reference value. In this case, any analog value between 0 and 5V is converted into its equivalent ADC value. 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 a maximal 10-bits ADC output value 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 the 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.

### 9.2.1 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 10: ADMUX Register.

The ADMUX Register (see Figure 10) is one of the control registers for the ADC. It holds the MUX bits, which select the channel (0 to 5 in our case, please note that these four bits provide a simple binary representation of the channel number), the reference selection bits - which specify the reference 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 11: 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 12: MUX settings

The main idea behind activating the ADLAR bit, is to use just the data from the 8-bit ADCH register (as two least significant bits, very often provide only noise) without any masking or other additional calculations.

The Figure 11 shows how to select the **REFS1** and **REFS0** bits. In the lab, we would use the  $AV_{CC}$  as the only reference, 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 12 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:

$$ADMUX |= (1 \ll MUX0) | (1 \ll MUX1);$$

To enable the value measurement, it is necessary to set the ADC Control and Status Register (ADCSRA). See Figure 13:

- **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. It 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.

|               |             |             |              |             |             |              |              |              |               |
|---------------|-------------|-------------|--------------|-------------|-------------|--------------|--------------|--------------|---------------|
| Bit           | 7           | 6           | 5            | 4           | 3           | 2            | 1            | 0            |               |
| (0x7A)        | <b>ADEN</b> | <b>ADSC</b> | <b>ADATE</b> | <b>ADIF</b> | <b>ADIE</b> | <b>ADPS2</b> | <b>ADPS1</b> | <b>ADPS0</b> | <b>ADCSRA</b> |
| 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 13: ADC Control and Status Registers.

**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 14: Selection of ADC Prescaler Bits.

- **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) will terminate the 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 (8 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  $F_{CPU} = 16\text{MHz}$ ,  $F_{ADC} = 16\text{M}/128 = 125\text{kHz}$ .

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 needs. There is a trade-off between the frequency and the 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 14.

Inside the hardware from the Atmel ATmega88PA is a built-in security lock. This lock prohibits reading an inconsistent value between the readings of the 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 the two registers in the correct order or better still, use the ADCW (this returns the whole 16-bit value and performs the whole calculation automatically for you):

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

### 9.2.2 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.

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

#### Task 12:

Write a program, which

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

**Hint:** On some boards it may be necessary to add a small delay (e.g.

```
_delay_ms(1);
```

) inside the infinite loop, as without it the MCU may hang up.

#### Task 13:

In order to use a “real” sensor: connect the photo sensor instead of the potentiometer to the input pin ADC3. Change the illumination condition of this sensor e.g. with your finger (or e.g. with your smartphone light). Is it possible to see all three LEDs on without additional light source? Describe it in your lab report. Disconnect the input pin ADC3 from the photo sensor and simply touch the wire connected to ADC3 with your finger. What happens? Describe it in your lab report and provide a short explanation.

**Task 14 (Optional):**

There are two files available in the template 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 show a string on the display.

Write a program that shows two ADC values (connected to the both potentiometers) on the LCD display, each in its own line. Feel free to select any two ADC channels (you will have to manipulate the MUX bits during the runtime). **Hint:** Check example programs, e.g. the tone generator already used the ADC value (on the LCD)!