HOCHSCHULE RHEIN-WAAL
Rhine-Waal University of Applied Sciences

LABORATORY

Microcontroller

1

EXPERIMENT:

Basic Microcontroller programming

1 Micro…

A few classes of controllers and systems:

- **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”. 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 a 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.

A few facts (selected) about this architecture:

- There are four sub-components in this architecture:
  - Memory:
    - for program code
    - for variables inside the program
  - Input/Output (called ”IO”)
  - Arithmetic-Logic Unit
  - Control Unit
- Often omitted: There is a 5th key player in this operation: a bus, or wire, 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.

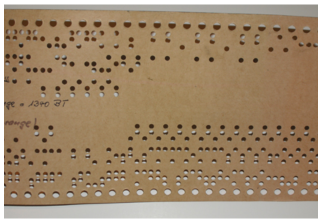


Figure 1: punch card

Figure 2: using a punch card

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 independent or dependent on a certain condition.

We can distinguish between several processors upon the maximum number of bits they can process in one step: (4 bit), 8 bit, 16 bit, 32 bit, 64 bit and so on. You might be used to 32 bit and 64 bit systems (eg. personal computers). But what about these bits? An 8 bit microcontroller can process 8 bits at one stroke. The biggest number, which can be expressed by 8 bits is 255.

$$max = 2^n - 1$$

This means: The 8 bit controller can for example 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 needs a (mostly) 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: Assigning jobs to several groups of students and expecting the result back on next Monday, to put it all together in a project. All the groups are working at a different speed, but the project is definitely done on Tuesday – synchronized. The maximum frequency of processors ranges between 1 MHz to more than 4 GHz (4 000 000 000 operations per second!) Microcontrollers are working 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) need more clock cycles to execute a single instruction. The instructions are more complex. RISC (Reduced Instruction Set Computer) use simpler (and less) instructions. The result is, that the processor often needs only one cycle to execute a single instruction. But so 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?

Three examples of devices with microcontrollers:

- Traffic lights
- TV remote
- MP3 player

2 Peripherals of a AVR microcontroller

Peripherals are parts of microcontrollers. Without the peripherals a microcontroller is 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 produce on- and off -signals, pulses, frequencies and read those kind of signals, too. But they are not able to read voltages in between those borders. For measuring e.g. 2,592 volts, or to compare a voltage to 3,214 Volts (bigger / smaller) the analog to digital converter or the analog comparator can 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 a 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 opponent of the A/D converter. It 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 a 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 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, Asynchron, 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 one 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 with e.g. an oscilloscope.
- **RAM:** Data can be written to and read from the Random Access Memory. RAM is volatile. The content will be lost / erased if the supply voltage is switched off. 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 of time 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. 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.
- **Flash:** A sort of EEPROM, but much faster. The number of writing actions is limited to 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 Block diagram

In this Lab we will use the ATmega88PA. The block diagram is in Figure 3. To connect the microcontroller to the other electronic parts, pins are used. The pins of the ATmega88PA are shown in Figure 4

2.2 Digital signals

Let's first have a look at the digital signals on the AVR. The digital signals (0 / 1, high / low) are voltages between 0 volts (ground) and the supply voltage, mostly 5 volts. The logic TRUE (1) always corresponds to the supply voltage. The CMOS inputs can source 3 to 10 mA and sink up to 20 mA. Almost every pin can be used as a digital I/O. See Fig. 3. There are special functions on every pin. They will be explained and dealt in the upcoming labs.

In this lab we will deal with the digital I/Os only. To be able to understand the internal processes, there is some information needed.

In Figure 5 is something most of you should have seen before: A PC's mainboard (or motherboard). There is a CPU (microprocessor), I/Os, memory, and busses on this board, too.

So basically all the components you see are present in our microcontroller. How do we transfer a program to the PC? Usually this is done by inserting a USB-flash, CD-ROM, by downloading a program from the Internet, or by just using the built in hard drive. That is the way we are used to.

2.3 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 programmer nowadays. At earlier times we had to build a programmer for the serial port (RS232) or for the parallel port. You can see an example in Figure 6.

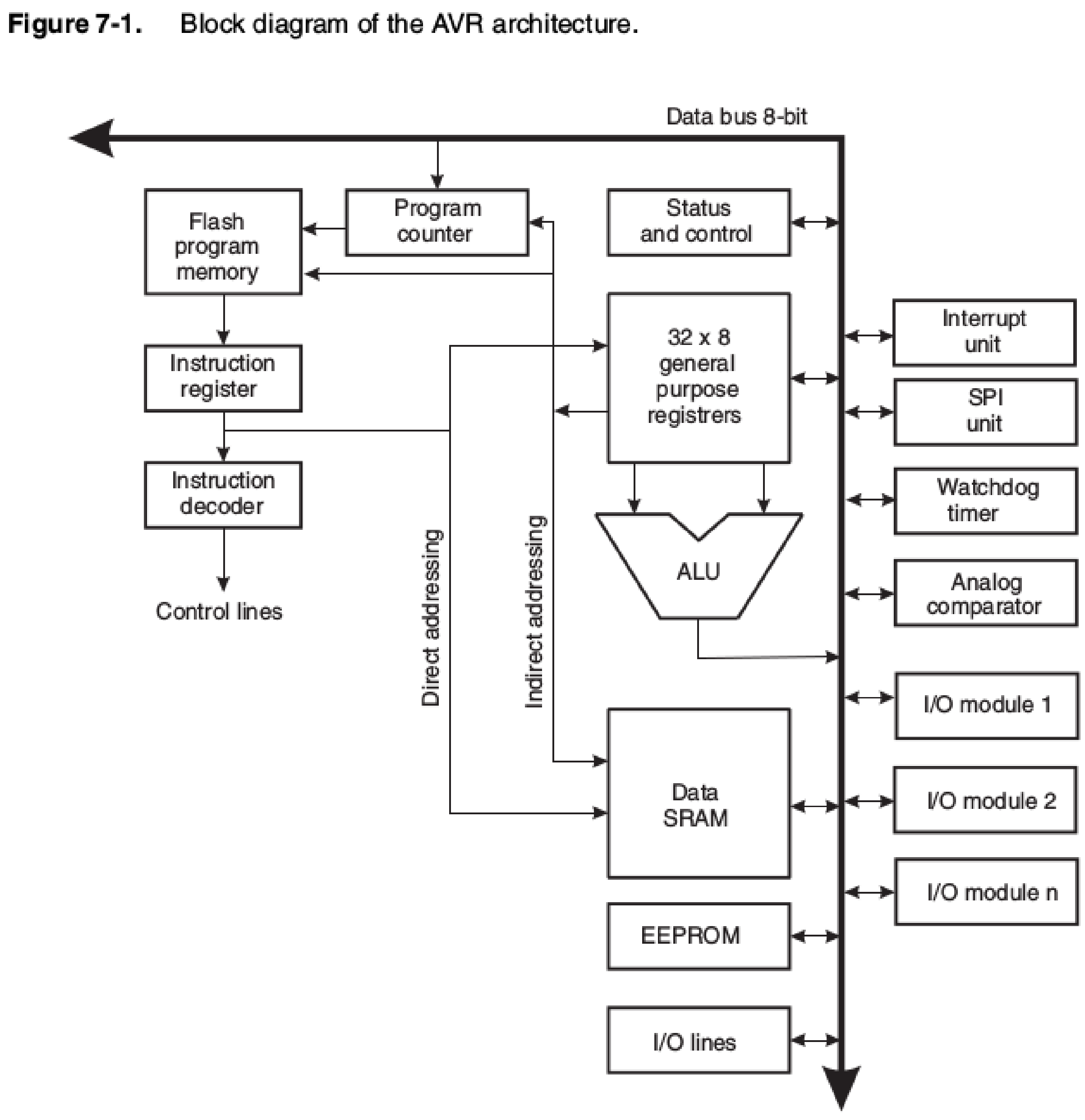


Figure 3: Block Diagram of the Atmel ATmega88PA

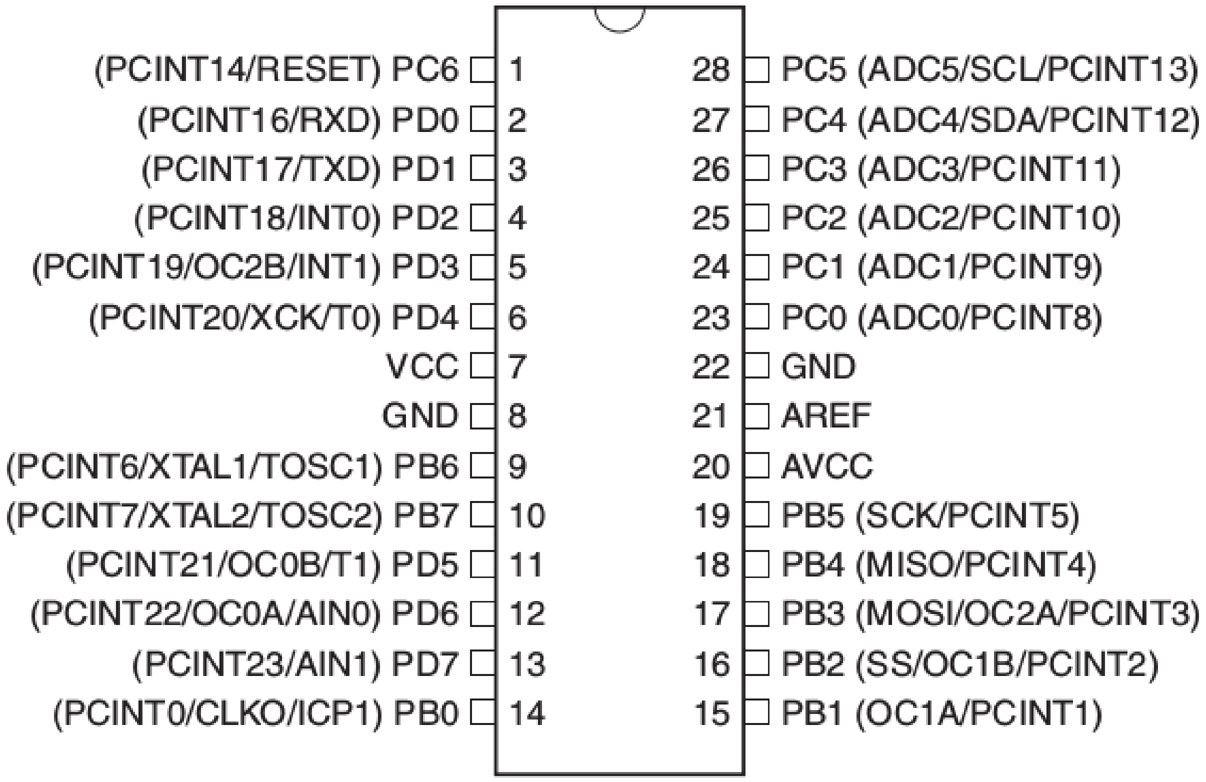


Figure 4: Pins of the Atmel ATmega88PA

Figure 5: A PC mainboard

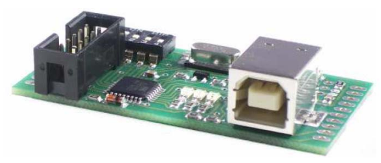


Figure 6: ISP with USB connector

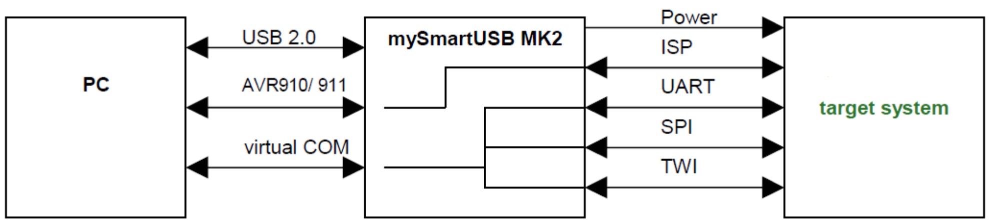


Figure 7: My Smart USB MK2 (ISP programmer)

This programmer is multifunctional. It provides a power supply and a serial interface to the controller as well (USB-serial bridge, or virtual COM). This means we are not only able to transfer a program to the microcontroller, we can ”talk” to it using the same USB connection on the computer! See Figure 7.

3 Program code

How does a program look like? Here is an example for a very simple program, which toggles an LED on Pin B-1, if a Button connected to D-2 is pressed:

```
:1000000012C024C023C022C021C020C01FC01EC0F7
:100010001DC01CC01BC01AC019C018C017C016C014
:1000200015C014C013C011241FBECFE5D4E0DEBF3D
:10003000CDBF10E0A0E6B0E0E8E9F0E002C0059036
:100040000D92A236B107D9F702D024C0D9CF1CD067
:10005000809160042E028EC30E001C080E08299AD
:100060000DC0882359F088B3842788BB80E293E0D1
:10007000F9013197F1F70197D9F7F0CF829BEFCFD4
:10008000882369F781E0EBCF8A988B98929A939AAC
:08009000B99A0895F894FFCF1E
:02009800010065
:00000001FF
```

first_c.hex

The file format is Intel hex. It contains the machine-readable instructions and initialization values. We will return to this file soon.

How does a program for a PC (application / software) look like? You can open the tarv14.exe. It is the executable file for the Target3001 Software.

```
é#“G§ ÐPæÝÕå6tëŸ
/êö, - Z... ¿ Ûm ûüa*f;‹k-—
w1ÄÂ8Â]áûPkå ?„nùm¿/...„ ©üo Ë§ÂS¡
Â,Â,A3-
```

snippet of targetv14.exe

The file format is different, but it still holds machine-readable instructions. In this case they are customized for the x86 processor architecture.

How is a computer-application programmed?

> For PCs there is no need to transfer the software you are developing to another system for testing. It can directly be executed (EXE file) on the source-system.

It is different for microcontrollers.

> The target is completely different to the source-system. And there is (mostly) no way to develop the software on the target itself. How should that be possible: There is no keyboard, no mouse, no CRT monitor and no operating system on the microcontroller system. Answer: We need to compile (translate) the source code into machine-readable code on a system, that itself cannot read the 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 "burn") the compiled program into the microcontroller

We are going to take a look at the "burn"-software. 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. We are going to put a precompiled software into it. Connect one button to PORTD / PIN 2, and one LED to PORTB / PIN 1. If you power up the board, nothing is going to happen. Now we will use avrdude to flash the first simpe test program "'program.hex"' to the microcontroller. The first step is to open a terminal. Inside this terminal you can enter commands.

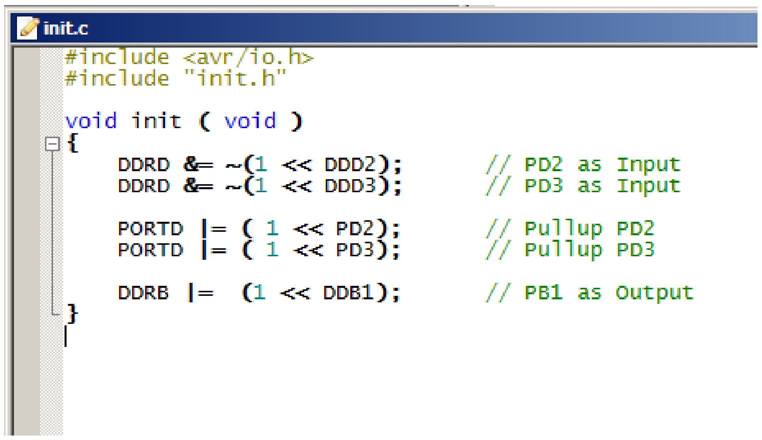
Task 1:
To open a terminal use the shortcut STRG + ALT + T. All files you need are inside a subdirectory. For all experiments there are pre written files (templates). To go into the directory with all templates enter the command: *cd Templates*. To change into the directory for this experiment enter *cd Experiment1*. This direcory contains the file program.hex.
Use:
avrdude -p m88p -c avr911 -P /dev/ttyUSB_MySmartUSB -U flash:w:program.hex:i
to put the file into the microcontroller.

Have a look at *avrdude –help*

- **-U flash** does something 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 successfull flashed, it possible to toggle the LED with the button.

Now you know the basics of how to transfer a compiled program to a microcontroller. But the compiled code is not human-readable. So what to do, if you want to use e.g. two buttons? A modification of the hex file is



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

void init ( void )
{
    DDRD &= ~(1 << DDD2);       // PD2 as Input
    DDRD &= ~(1 << DDD3);       // PD3 as Input

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

    DDRB |=  (1 << DDB1);       // PB1 as Output
}
```

Figure 8: File init.c

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 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 program.hex, see Figure 8 and Figure 9.

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 too much about the code around. But what you should understand are the bitwise logic operations like:

&, ˆ, ~

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

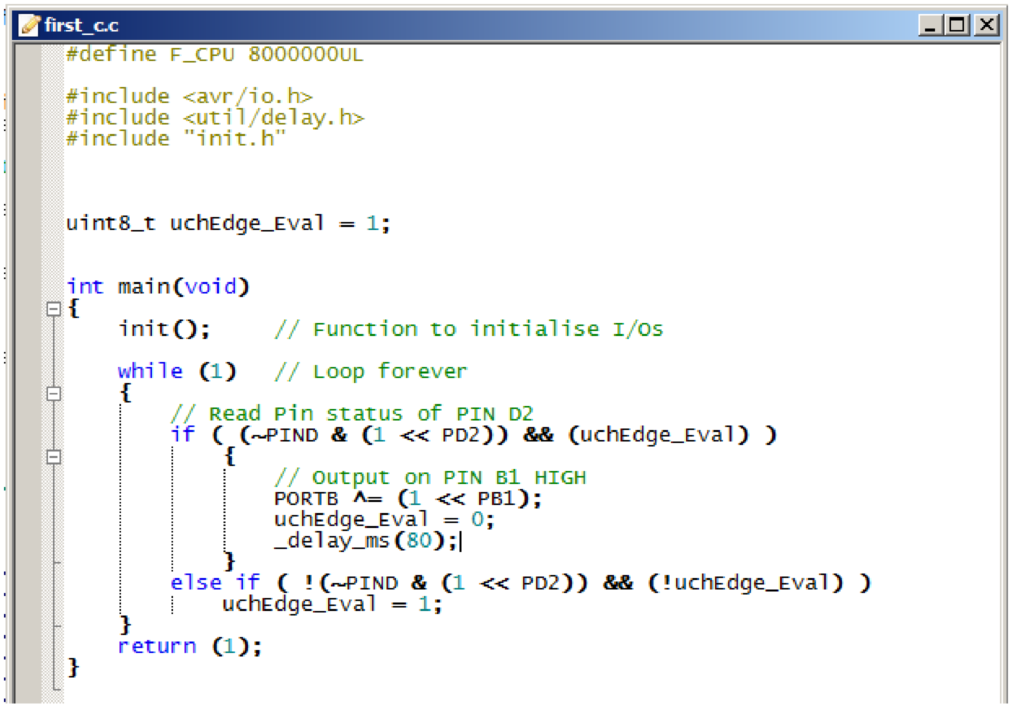
3.2 Assembler language

But for now, we will enter the programming one step “below” C, by using assembler language, see Figure 10.

Let's open the ”assembler_output.lss” to have a look at the assembler code. This file holds assembler code, which was generated by the compiler (see Figure 10). The compiler converts the C-file into assembler code.

Let's have a look at the init function, see Figure 11.

Compare these assembler instructions with the instruction set summary and the register summary. The block diagram might be useful, too.



```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 9: File first_c.c

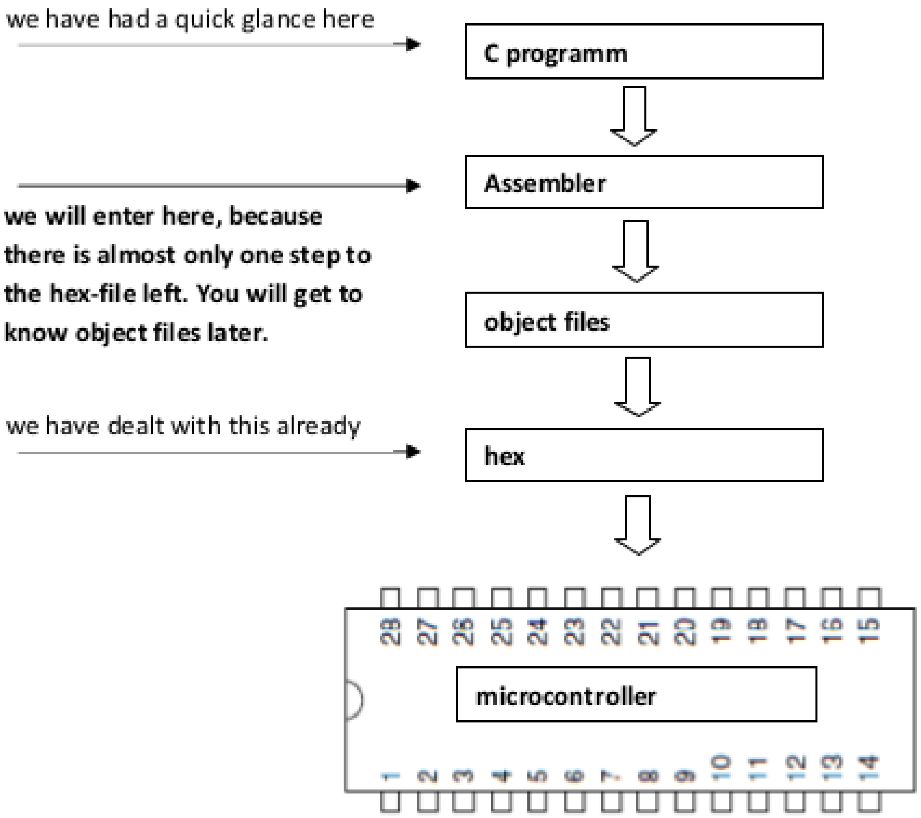
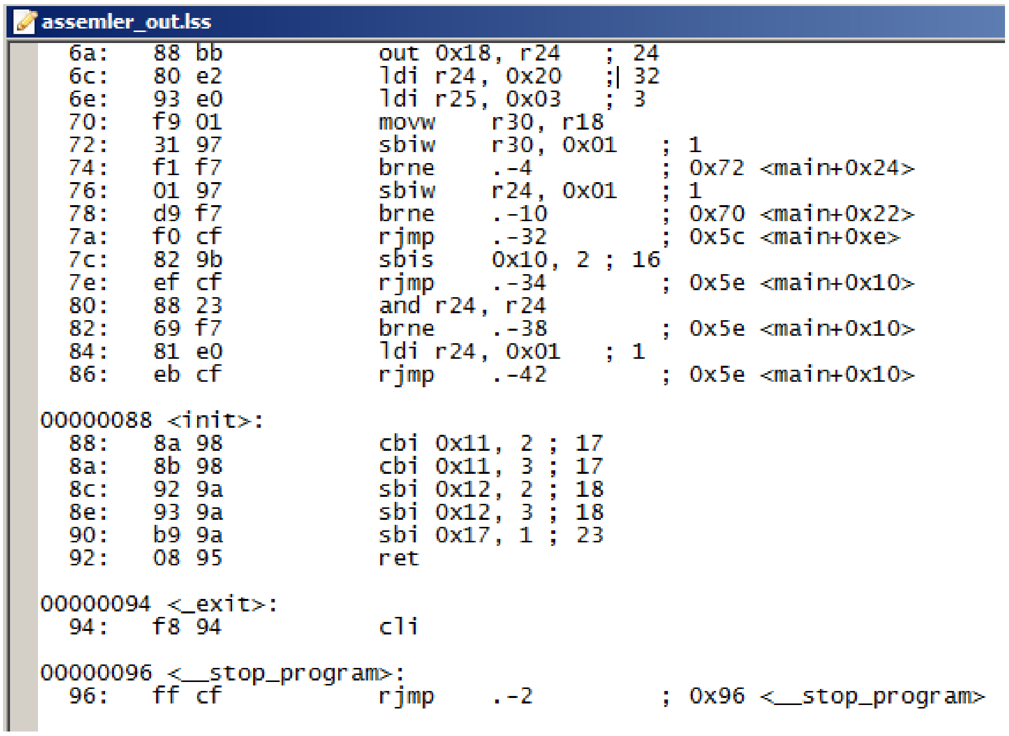

we have had a quick glance here
C programm
Assembler
we will enter here, because there is almost only one step to the hex-file left. You will get to know object files later.
object files
we have dealt with this already
hex
28 27 26 25 24 23 22 21 20 19 18 17 16 15
microcontroller
1 2 3 4 5 6 7 8 9 10 11 12 13 14

Figure 10: Programming steps



```
 6a:   88 bb          out 0x18, r24   ; 24
 6c:   80 e2          ldi r24, 0x20   ; 32
 6e:   93 e0          ldi r25, 0x03   ; 3
 70:   f9 01          movw    r30, r18
 72:   31 97          sbiw    r30, 0x01    ; 1
 74:   f1 f7          brne    .-4          ; 0x72 <main+0x24>
 76:   01 97          sbiw    r24, 0x01    ; 1
 78:   d9 f7          brne    .-10         ; 0x70 <main+0x22>
 7a:   f0 cf          rjmp    .-32         ; 0x5c <main+0xe>
 7c:   82 9b          sbis    0x10, 2 ; 16
 7e:   ef cf          rjmp    .-34         ; 0x5e <main+0x10>
 80:   88 23          and r24, r24
 82:   69 f7          brne    .-38         ; 0x5e <main+0x10>
 84:   81 e0          ldi r24, 0x01   ; 1
 86:   eb cf          rjmp    .-42         ; 0x5e <main+0x10>

00000088 <init>:
 88:   8a 98          cbi 0x11, 2 ; 17
 8a:   8b 98          cbi 0x11, 3 ; 17
 8c:   92 9a          sbi 0x12, 2 ; 18
 8e:   93 9a          sbi 0x12, 3 ; 18
 90:   b9 9a          sbi 0x17, 1 ; 23
 92:   08 95          ret

00000094 <_exit>:
 94:   f8 94          cli

00000096 <__stop_program>:
 96:   ff cf          rjmp    .-2          ; 0x96 <__stop_program>
```

Figure 11: assembler code

Let's now look at our hex-file:

```
:1000000012C024C023C022C021C020C01FC01EC0F7
:100010001DC01CC01BC01AC019C018C017C016C014
:1000200015C014C013C011241FBECFE5D4E0DEBF3D
:10003000CDBF10E0A0E6B0E0E8E9F0E002C0059036
:100040000D92A236B107D9F702D024C0D9CF1CD067
:10005000809160042E028EC30E001C080E08299AD
:100060000DC0882359F088B3842788BB80E293E0D1
:10007000F9013197F1F70197D9F7F0CF829BEFCFD4
:1000800088 2369F781E0EBCF8A988B98929A939AAC
:08009000B99A0895F894FFCF1E
:02009800010065
:00000001FF

                                   first_c.hex
```

3.3 Summary

Now you basically know how a human-readable program is translated (via a few steps) to machine code. Next time we will have a closer look at the functions of registers, and we will start writing a first simple program in assembler language. Make sure, you have a rough understanding of the processors architecture, the registers (register summary) and the instructions (we will only need the ldi, out and sbi instruction).

4 Example programs

The downside of the MYAVR board, it has no isolation. Never place this board on conductive things (like a pen or metal). The table in the lab is not conductive.

Not let us check four simple programs. You can find an example how to connect in Figure 12. Before flashing and running the programs connect the microcontroller with the actors and sensors:

- 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 poti 1
- Pin C3 with the sounder (on some boards labbeld with "’summer"’)

Task 2:
After this is connected, flash the example programs and test the result.

The commands to flash a program are:

- *make flash1*
- *make flash2*
- *make flash3*
- **...**

4.1 Flip Flop

Program name: **one**

This programs enables or disables the LEDs, controlled by the keys.

- Key 1: all LEDs on
- Key 2: all LEDs off
- Poti: no function

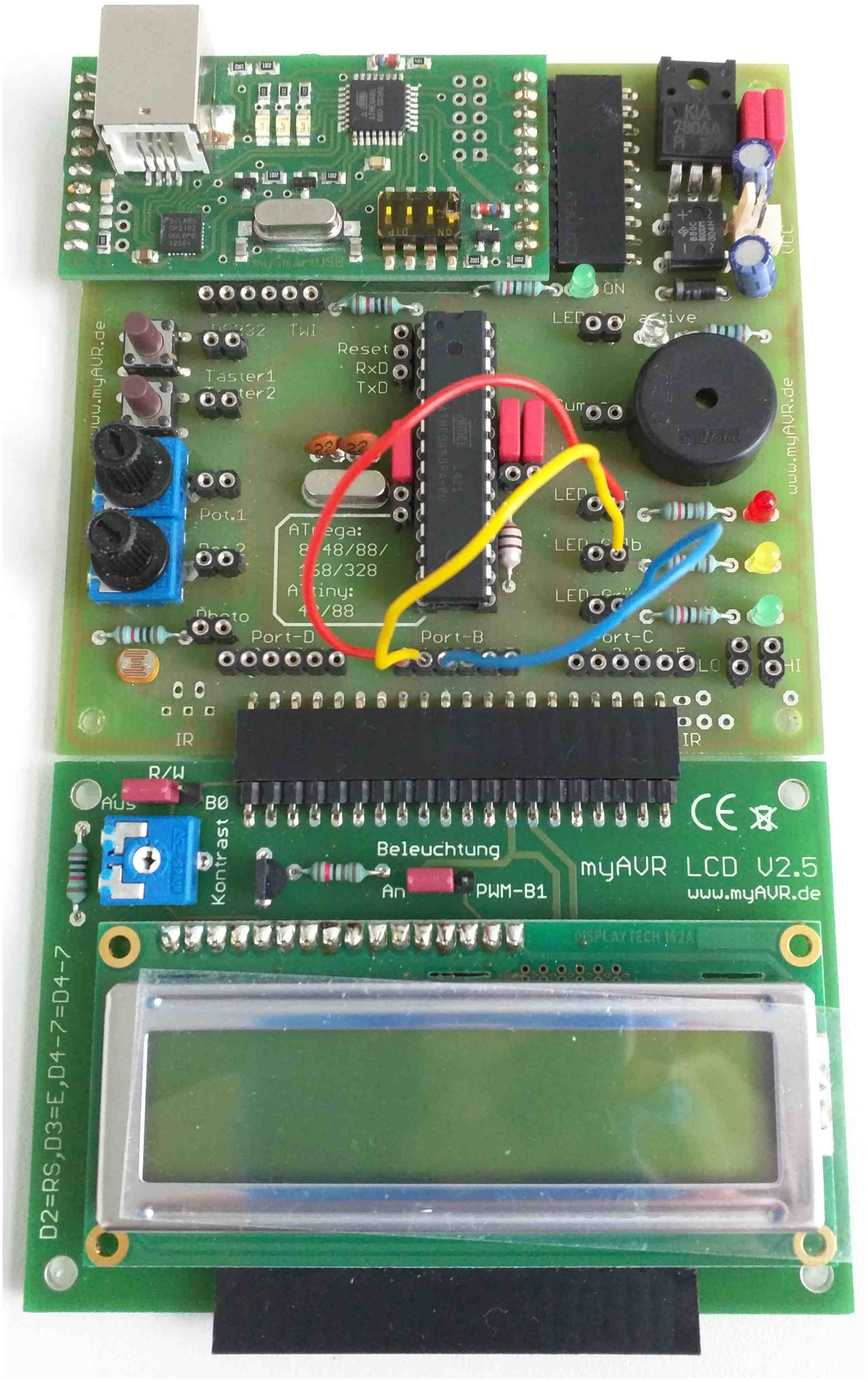


Figure 12: Example for connections on the board

4.2 Traffic light

Program name: **two**

This programs simulates a simple traffic light.

- Key 1: no function
- Key 2: no function
- Poti: no function

4.3 Tone generator

Program name: **three**

- Key 1: A tone will be generated by holding the key.
- Key 2: Another tone will be generated by holding the key
- Poti: no function

4.4 Music playing

Program name: **four**

- Key 1: Press to play a little bit music
- Key 2: no function
- Poti: no function

4.5 Moving Display Output

Program name: **five**

- Key 1: Press to re-play
- Key 2: Press to re-play
- Poti: no function

4.6 Function Generator

Program name: **six**

This program generates a a square wave signal with a duty cycle of 50%.

- Key 1: no function
- Key 2: no function
- Poti: set the output frequency

5 C programming

We are going to start with a simple program. Change into the subdirectory 'mycode' with the *cd* command. Open the file as "main.c", see Figure 13.

Hint: Here is a schematic about the behavior of the data direction registers (DDR), the PORT and the PIN registers. If we write something to the PORT register, only those "lines" will appear on the output (the actual terminal to the outside), whose corresponding bits in the DDR are set to 1 (marked for "output"). Vice versa: if we perform a "read" command, only those input lines whose corresponding bits are set to zero in the DDR, will appear in the register, see Figure 14.

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

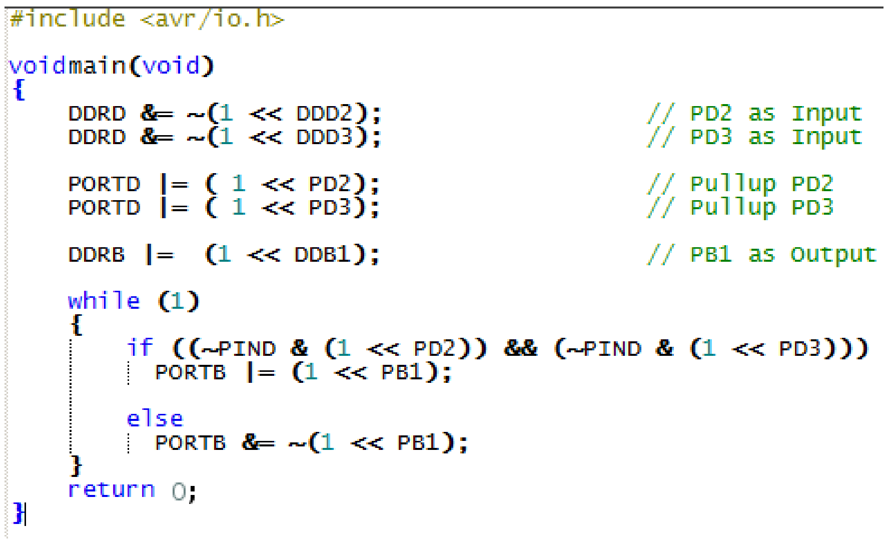
There you can find operations like: "~", "&", "&&" and "<<"

- "~" (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 (left in this case) operation.

Bytes can represent a logical value. A byte that is zero is **false**. All other is **true**. Logical operations have only a logical result (true/false).

Prepare 1: What are the results for (A=10011110, B=00101101): bitwise negation of A, A & B, A && B, A || B, A | B?

Hint: The buttons on this board are only able to establish a connection to ground. If we simply connect it to the microcontroller, it turns out to be the situation like in Figure 15 and Figure 16.



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

voidmain(void)
{
    DDRD &= ~(1 << DDD2);                          // PD2 as Input
    DDRD &= ~(1 << DDD3);                          // PD3 as Input

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

    DDRB |=  (1 << DDB1);                          // PB1 as Output

    while (1)
    {
        if ((~PIND & (1 << PD2)) && (~PIND & (1 << PD3)))
          PORTB |= (1 << PB1);

        else
          PORTB &= ~(1 << PB1);
    }
    return 0;
}
```

Figure 13: A simple C program

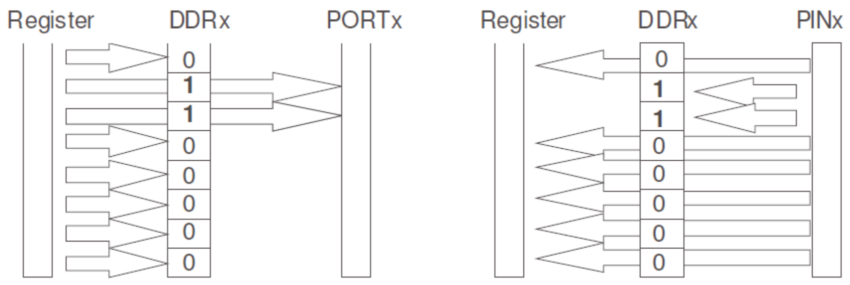
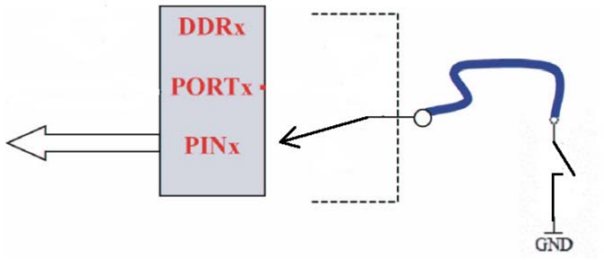
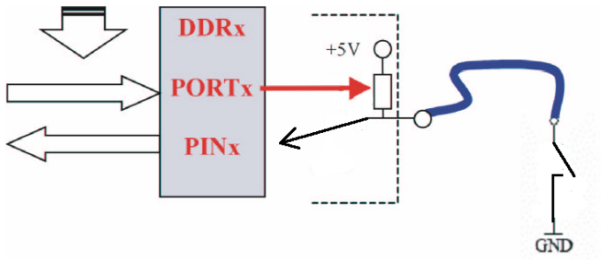


Figure 14: Data Direction Register



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 16.

Figure 15: 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 16: Button, internal pullup is 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/mycode). Type “ls” to see all files in a directory and use “cd” to change the directory. Compile the .c file using the previously introduces command. Type “ls” again. Which file(s) have been created now? (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 for the output
- **main.o** filename for the input

Task 4:
Open the command prompt. Link the .c file using the command above. Which file(s) have been created now?

Task 5:
Generate the hex file (ihex format).

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 [options] in-file [out-file]
```

```
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 copy 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 on the PC. We only change the file format. As soon as you finished executing this command, there should be the HEX file in the folder.

This method is obviously 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 has to be a possibility to automate this 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)
- 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 "init();". Now execute this program on the microcontroller.

To execute the program on the microcontroller you need to do this 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. 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! This PDF file is generated with LaTeX so be careful with copy and paste operations.

```
all:  main.hex

main.hex:  main.elf
	avr-objcopy  -O  ihex  main.elf  main.hex

main.elf:  main.o  init.o
	avr-gcc  -mmcu=atmega88pa  -o  main.elf  main.o  init.o

main.o:  main.c
	avr-gcc  -mmcu=atmega88pa  -c  -Os  main.c

init.o:  init.c  init.h
	avr-gcc  -mmcu=atmega88pa  -c  -Os  init.c

program:  main.hex
	avrdude  -P  /dev/ttyUSB_MySmartUSB  -p  m88p  -c  avr911  -Uflash:w:main.hex:i

clean:
	rm -f main.hex
	rm -f *.o *.elf *~
```

Task 7:
Complete the following tasks:

1. Write the makefile, run the automated compilation by typing ”make all” on the command prompt (navigate to the working directory fist).
2. Flash the hex file (flash means: program the controller with the HEX-file.).
3. Invert the behaviour of the LED, compile the program again and flash it onto the controller.
4. If that works fine add a second LED, which indicates that either button one or button two is pressed.
5. Recompile and flash.
6. Use the third LED, this LED should represent the XOR function for the two buttons. In C there is no logical XOR operator. To get an logical "A xor B" you can use: "!A != !B".

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 this functions in you program. The datatype **uint8_t** is here used as an **bool** (logical value). Zero is false and any other value is true.