HOCHSCHULE
RHEIN-WAAL
Rhine-Waal University
of Applied Sciences

LABORATORY

Microcontroller

1

EXPERIMENT:

Basic Microcontroller programming

Please read the whole document before starting the experiment. This document contains 9 preparation jobs. It is mandatory to make all preparation jobs in written form! Without a written document that includes all preparation jobs it is not possible to pass this experiment.

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.

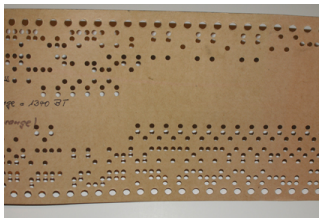


Figure 1: punch card

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

Prepare 1: Do some research and find out, which other devices might contain -or even be controlled by a microcontroller nowadays (minimum: 3).

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".

  Prepare 2: What does 10 bit A/D converter mean? If the controller uses 5 volts supply and this is also the maximum input voltage for the A/D converter, how big is the resolution of the A/D converter in volts?
- **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.

Prepare 3: Do some research on serial communication, RS232 and be able to explain the method of serial communication. What are the voltage levels for the bits?

- **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.

Figure 7-1. Block diagram of the AVR architecture.

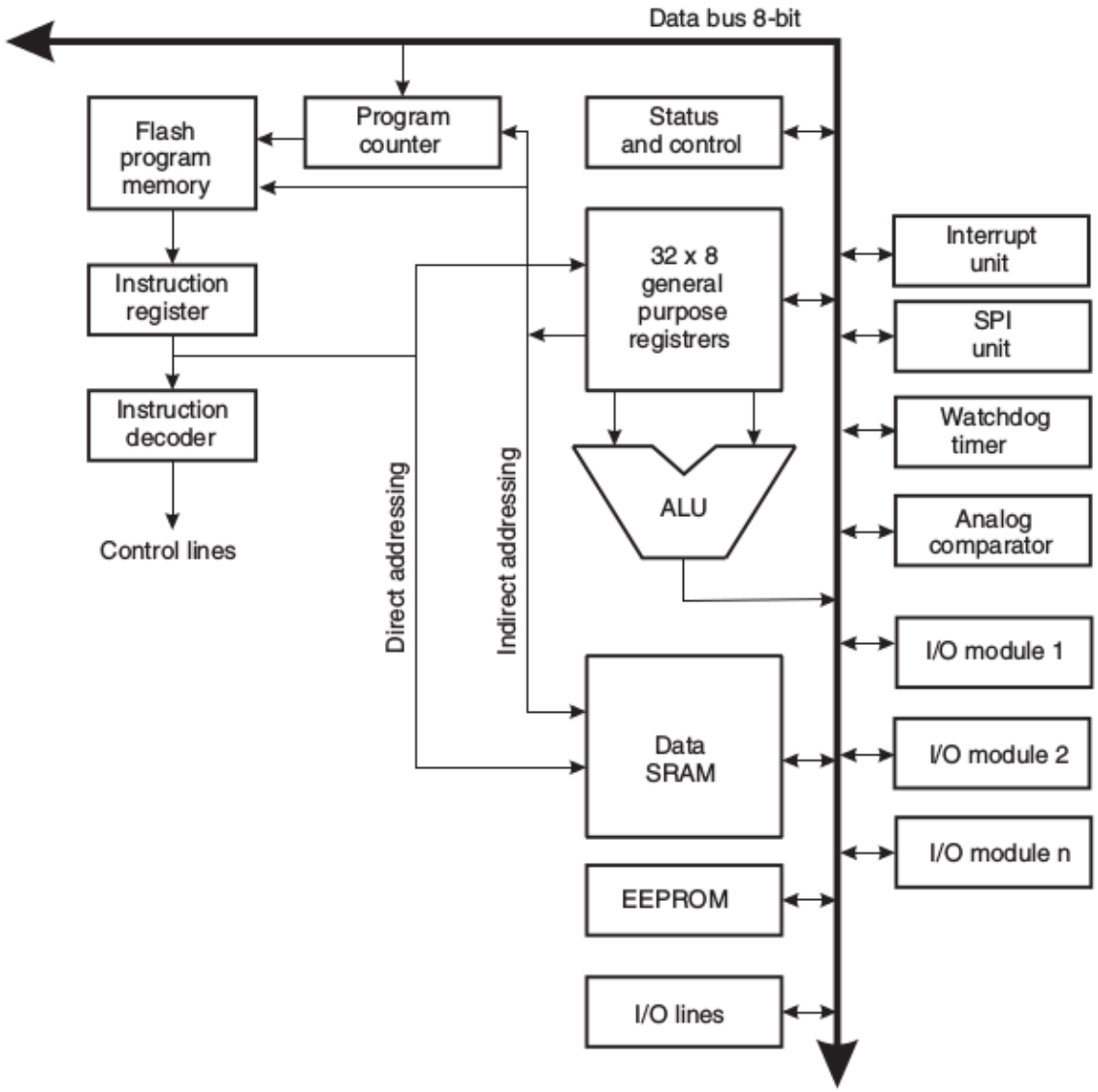


Figure 3: Block Diagram of the Atmel ATmega88PA

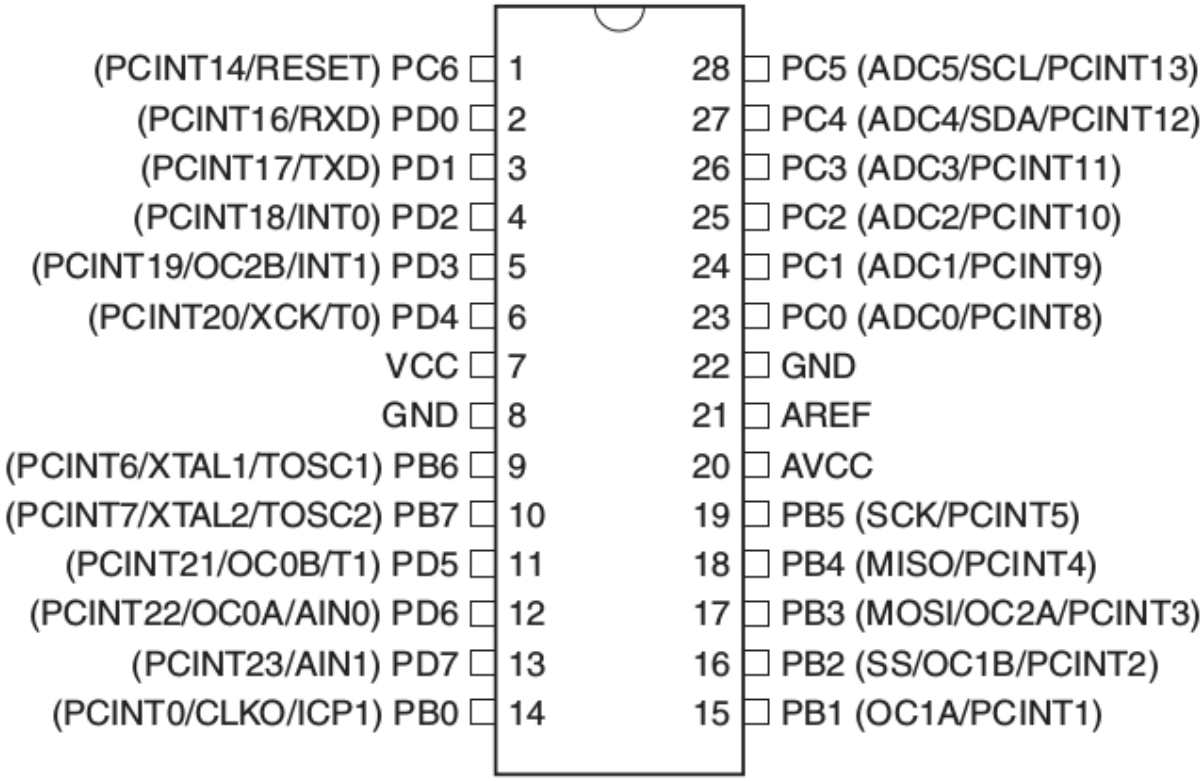


Figure 4: Pins of the Atmel ATmega88PA

Figure 5: A PC mainboard

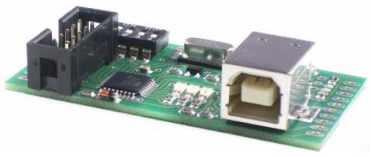


Figure 6: ISP with USB connector

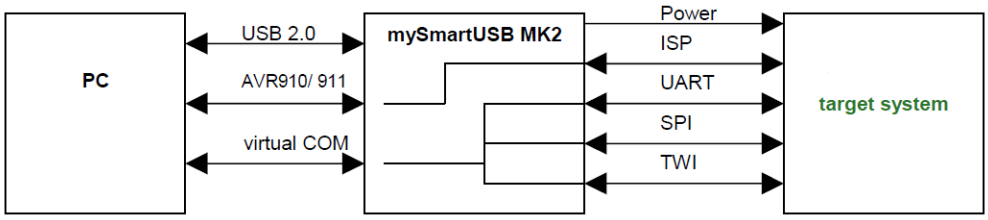


Figure 7: My Smart USB MK2 (ISP programmer)

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.

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
:1000500008091600042E028EC30E001C080E08299AD
: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.

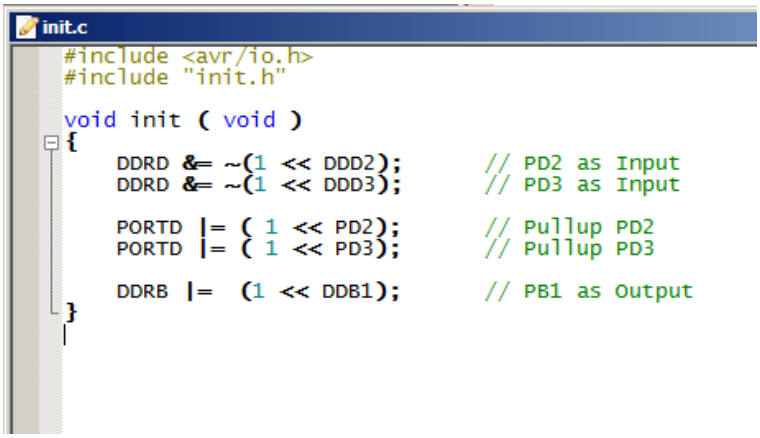


Figure 8: File init.c

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. 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 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.

first_c.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

Do not bother too much about the code around. But what you should understand are the bitwise logic operations like:

&, ^, ~

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.

Prepare 4: What is cbi and sbi in the lines 88 to 90?

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

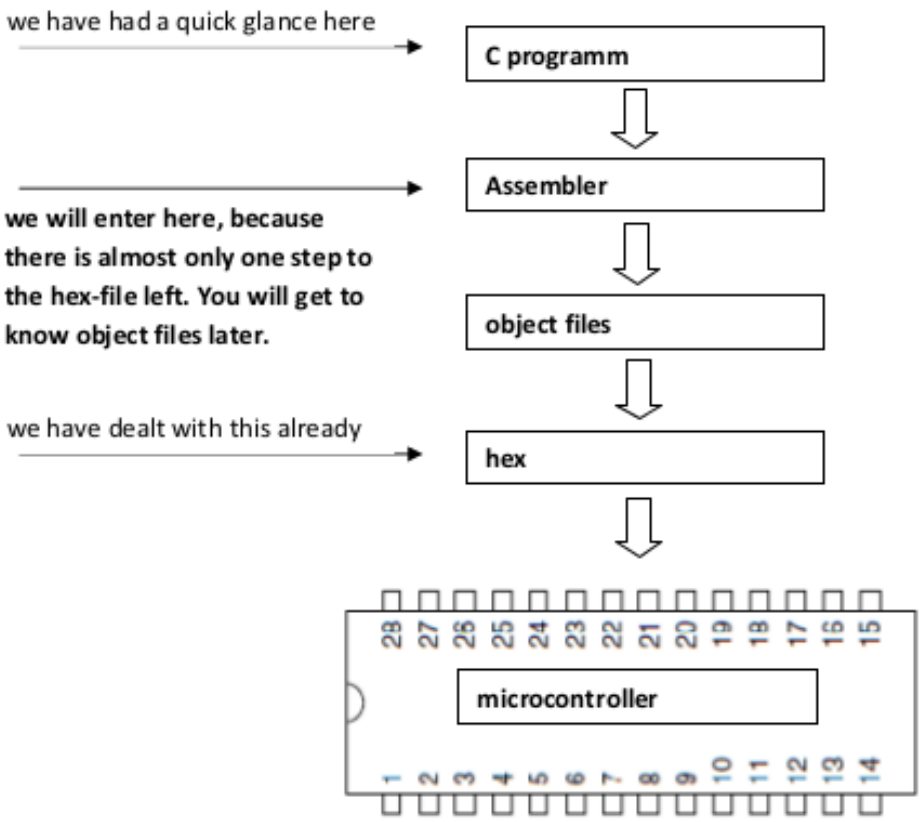


Figure 10: Programming steps

assemler_out.lss

```
   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
:1000500080916000 42E028EC30E001C080E08299AD
:100060000DC0882359F088B3842788BB80E293E0D1
:10007000F9013197F1F70197D9F7F0CF829BEFCFD4
:10008000882369F781E0EBCF8A988B98929A939AAC
: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 C5 with the sounder (on some boards labbeld with "'summer"')

After this is connected, flash the example programs and test the result. The commands to flash a program are:

- *make flash1*

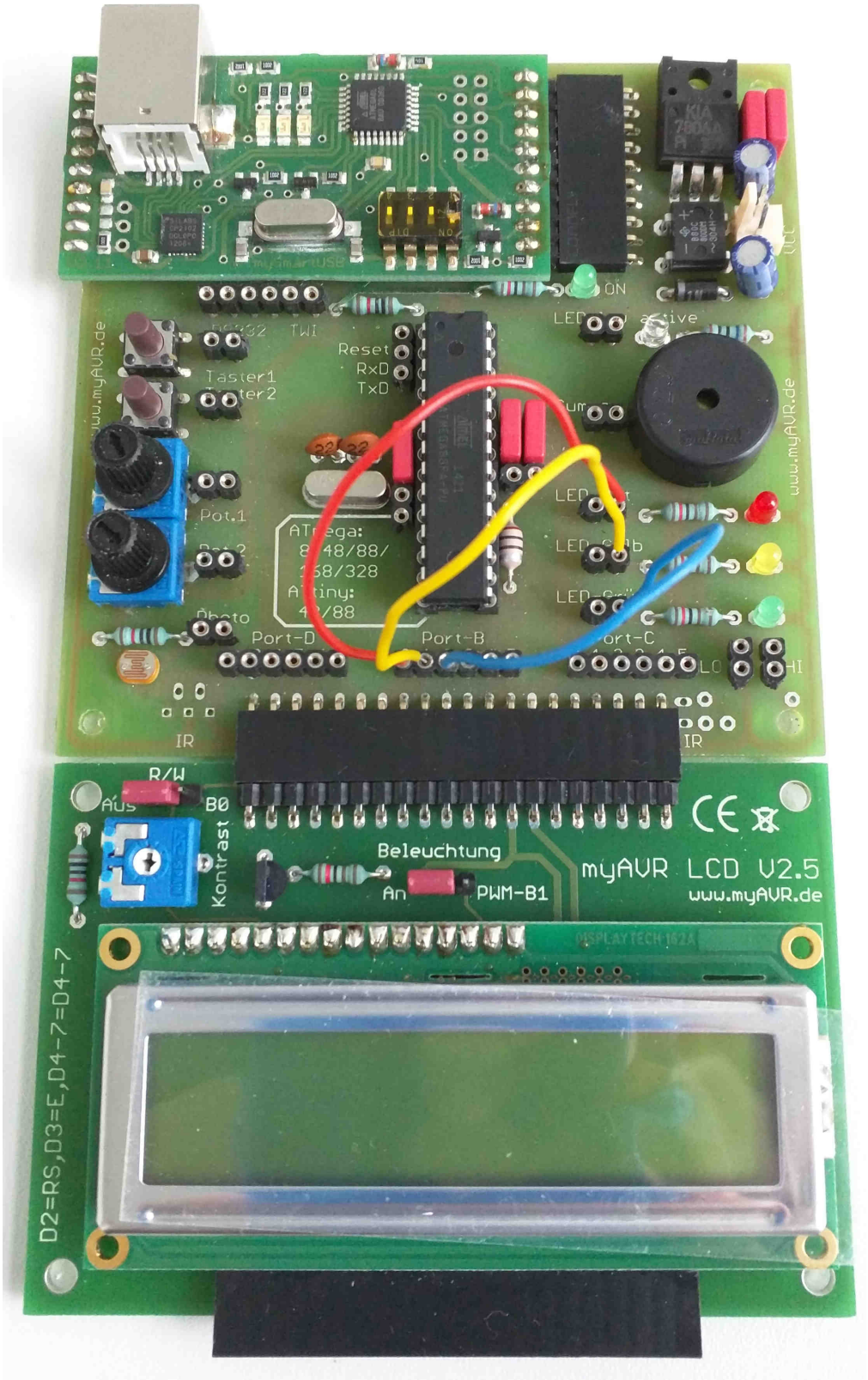


Figure 12: Example for connections on the board

- *make flash2*
- *make flash3*
- *make flash4*

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

4.2 Traffic light

Program name: **two**

This programs simulates a simple traffic light.

- Key 1: no function
- Key 2: 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

4.4 Music playing

Program name: **four**

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

```c
#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

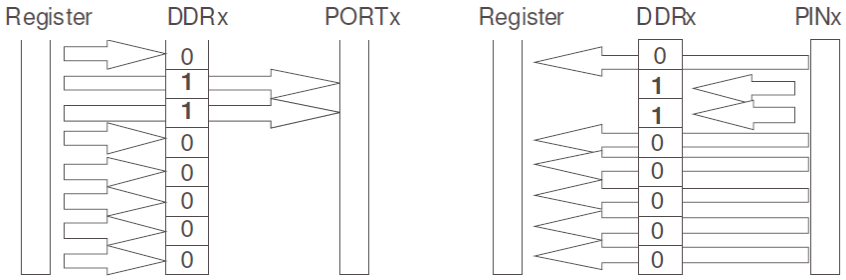


Figure 14: Data Direction Register

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.

Prepare 5: Understand the program and be able to connect LEDs and buttons to the correct pins, according to the program code. What does this program do? When is the LED on?

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 "<<"

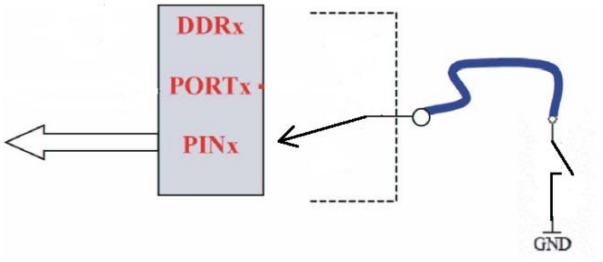
- "~" (called tilde) is a bitwise negation
- "&" is a **bitwise** and operation (bit by bit)

- "&&" 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).

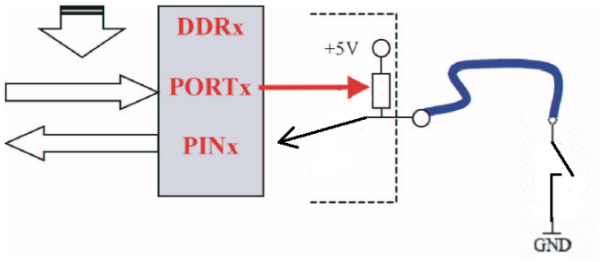
Prepare 6: What are the results for (A=10011110, B=00101101): bitwise negation of A, A & B, A && B, A || B, A | B?

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.



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

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

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

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”

Prepare 7: What does ”include guard” mean?

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:

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.

```
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 *~
```

Prepare 8: Understand this simple makefile. What is the command to build and write the program into the microcontroller in one step?

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".

Prepare 9: What is the truth table for XOR?

If all at this point is complete and works fine, inform the staff. You must be able to show and explain all steps you have done. The next steps are optional.

7 Modular programming

This section is 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.

HOCHSCHULE
RHEIN-WAAL
Rhine-Waal University
of Applied Sciences

LABORATORY

Microcontroller

2

EXPERIMENT:

Input and Output

Please read the whole document before starting the experiment. This document contains 4 preparation jobs. It is mandatory to make all preparation jobs in written form! Without a written document that includes all preparation jobs it is not possible to pass this experiment.

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. 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

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.

Prepare 1: What is a debouncing circuit?

Prepare 2: How to use the ”delay.h” with the AVR-GCC compiler to wait 10ms?

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;
```

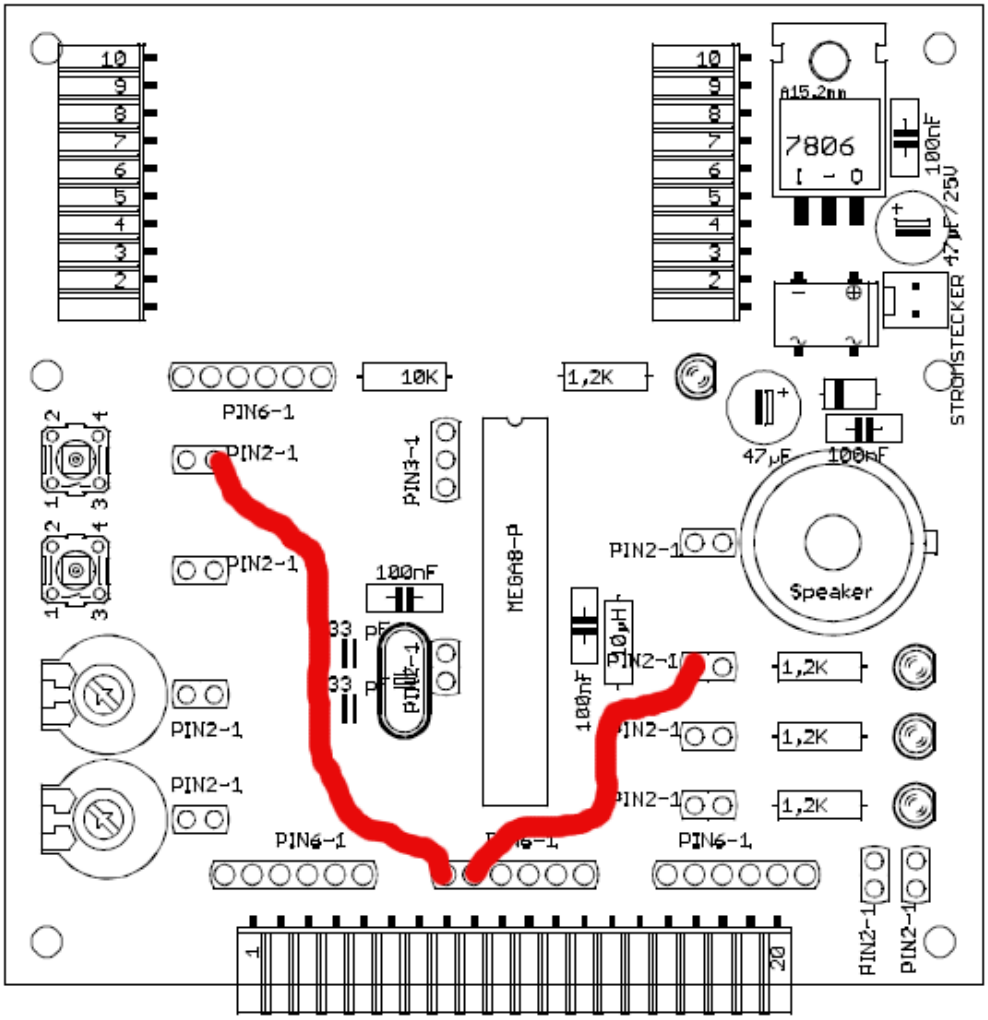
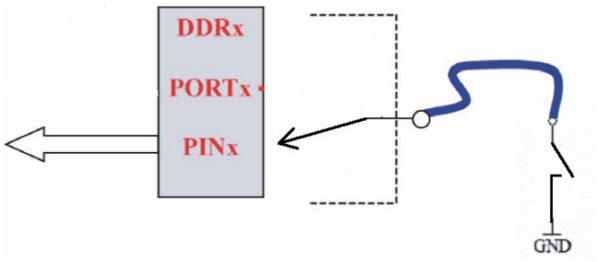
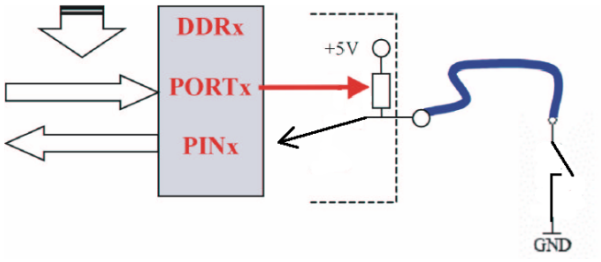


Figure 1: Simple connection with one button and one LED



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.

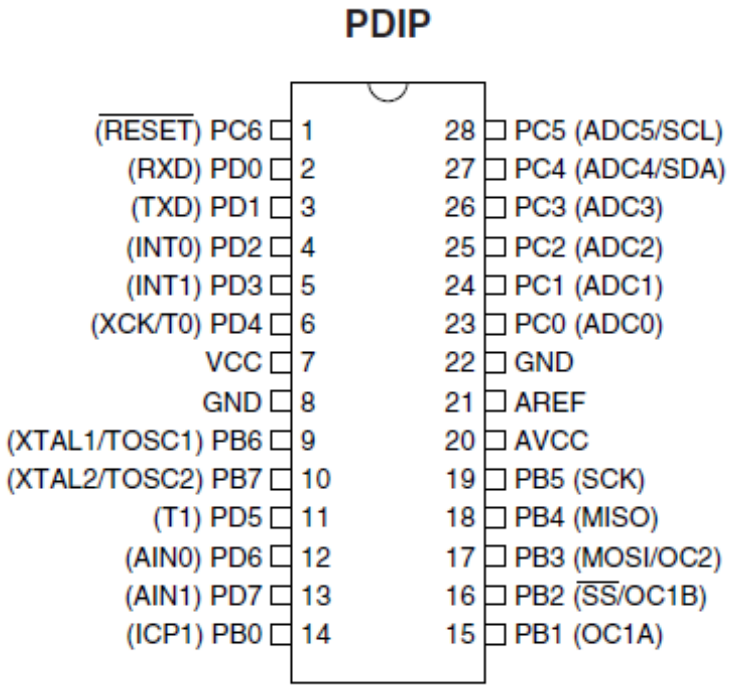
2.1 Analog output

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

Prepare 3: What electronic circuit is necessary to change a PWM signal into an analog signal?

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.

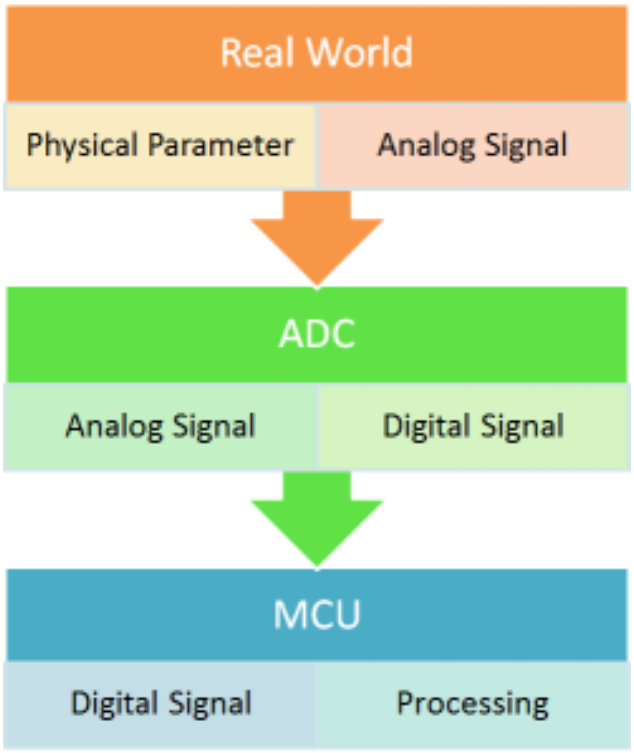


Figure 5: Signal Acquisition Process

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

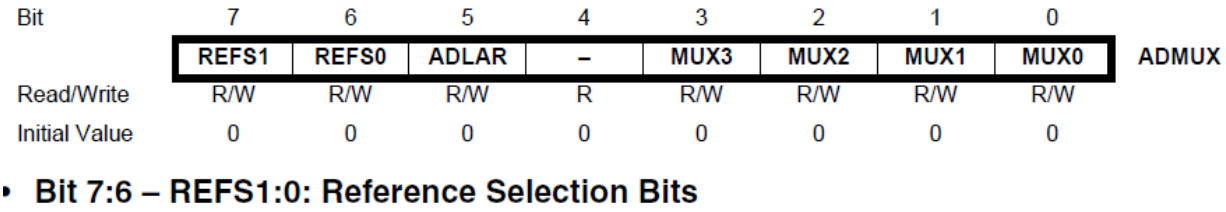
Suppose we use a 5V reference. In this case, any analog value in between 0 and 5V is converted into its equivalent ADC value as shown in Figure 5. The 0-5V range is divided into $2^{10} = 1024$ steps. Thus, a 0V input will give an ADC output of 0, 5V input will give an ADC output of 1023, whereas a 2.5V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

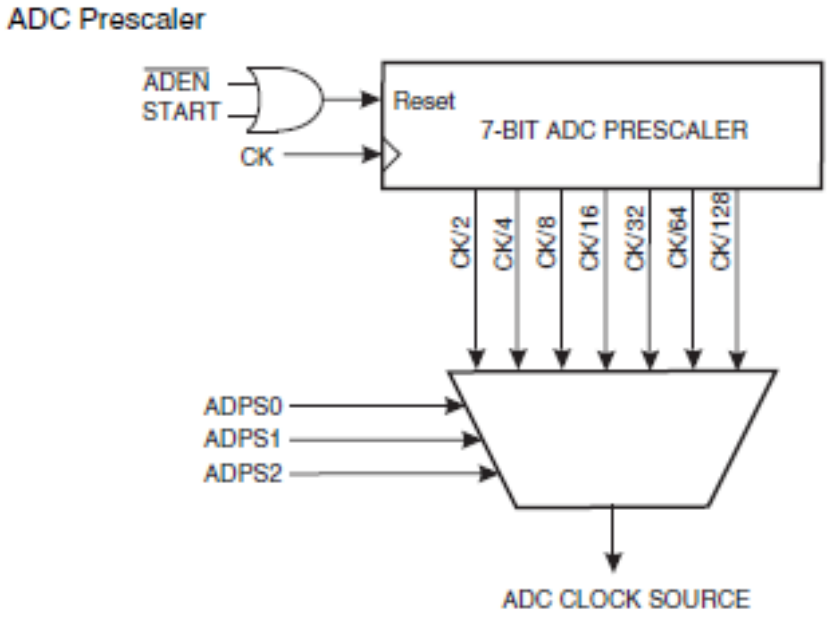


Figure 10: ADC Prescaler

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. 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

Prepare 4: Calculate the digital values for the voltage borders.

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

If all at this point is complete and works fine, inform the staff. You must be able to show and explain all steps you have done. The next steps are 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.

HOCHSCHULE
RHEIN-WAAL
Rhine-Waal University
of Applied Sciences

LABORATORY

Microcontroller

3

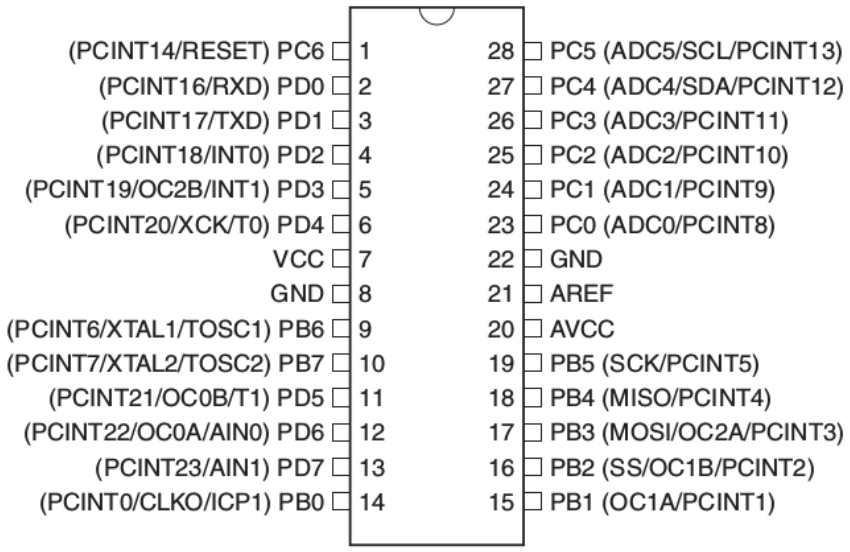
EXPERIMENT:

Timer and Counter

Please read the whole document before starting the experiment. This document contains 3 preparation jobs. It is mandatory to make all preparation jobs in written form! Without a written document that includes all preparation jobs it is not possible to pass this experiment.

1 System clock

The processor on the MyAVR board uses an external clock source of 8 MHz (it's a crystal oscillator, connected to the pins 9 and 10, see Figure 1). This clock is used to synchronize the internal processes. All microcontrollers use clocks. A basic instruction is processed when a tick from the clock passes. Just like these programs we are writing, as clock ticks pass, instructions are processed in time with the ticks of the clock.



External crystal oscillator is connected to PIN 9 and 10.

Figure 1: Atmel Mega88PA Microcontroller

2 Timer and Counter

The timer and counter functions in the microcontroller simply count in sync with the microcontroller clock. There are 3 Timers/counter in the AtMega88PA. Timer0 and Timer2 can only count up to 255 (8-bit counter), Timer1 can count up to 65535 (16-bit counter). That's far from the 8000000 ticks per second that the AVR on our board provides. The microcontroller provides a very useful feature called prescaling. Prescaling is simply a way for the counter to skip a certain number of microcontroller clock ticks. The AVR microcontrollers allow prescaling numbers of: 1, 8, 64, 256 and 1024. If –for example- 64 is set as the prescaler, then the counter will only count every time the clock ticks 64 times. That means, in one second (where the microcontroller ticks 8000000 million times) the counter would only count up to 125000.

Timers have a register for control and another register that holds the count number. The control register contains some switches to turn on and off features. One of the features is which prescaling to select. The control registers are called TCCR0A and TCCR0B (for Timer0), TCCR1A and TCCR1B (for Timer1). Here's a snippet from the datasheet (Page 133 and 134), see Figure 2 and Figure 3. The bits 0 to 2 determine the Timer1 prescaler.

The TCCR0 is the 8-bit control register for Timer0. There are only 8 switches to turn on and off. TCCR1 is 16-bit, so it has 16 switches to turn on and off, but it comes in two 8-bit registers labeled A and B (TCCR1A and TCCR1B). The register, that holds the count is called the TCNT register. And there is an 8-bit version (TCNT0) and a 16-bit version (TCNT1). The TCNT1 register actually gets its number from two other 8-bit

Timer/Counter 1 Control Register B – TCCR1B

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

Figure 2: Timer/Counter1 Control Register B

Table 16-5. Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

Figure 3: Clock Select Bit Description

registers (TCNT1H and TCNT1L, Low and High byte) to create a complete 16-bit number. If we want to stick to the example and turn on the Timer1 with a prescaler of 64, we would set CS11 and CS10 in the TCCR1B register to 1. This is done in the same way, like setting bits in the PORT register (without affecting the other bits), using the logic OR (bitwise OR) operator |. The counter value (which can be read by using "TCNT1") would then be incremented every 8 microseconds, Figure 4 explains this visually. We can not only read the counter value in TCNT, we can also write (e.g. setting it back to zero).

Prepare 1: You can find the necessary information about timers in the section "8-bit Timer/Counter0" in the datasheet. Please have a look and note the page numbers.

2.1 Flashing LED

Use the template files (in "Templates/Experiment3"), modify the init function in such a way, that Pin 0 on PortB is used as the only output. There will be no input necessary, at the moment.

Connect an LED to this pin (Figure 5) and make it blink every second by using Timer1.

Prepare 2: Find out a good value for the prescaler and think of a way how to generate a beat of 1 Hz. Some more mathematical expressions and IF statements might be necessary. There are multiple solutions. The best solution is the lowest possible prescaler, this has the best precision.

For this, you can use a simple compare (if counter value is equal or greater to.... do something), to trigger the action on the outputs.

Prepare 3: Why would you need a >=, and not only a == compare in this case?

2.2 Audio output

Generate a 300 Hz signal (50% duty cycle) using the same setup (Timer1, LED). Maybe it is necessary to change the prescaler. The LED should now be 50% dimmer than before. You are applying a 300 Hz software PWM signal (with fixed 50% duty cycle) to the LED, see Figure 6. The Atmel Mega88PA has an internal hardware to generate PWM signals, do not use this here. The LED only appears to be dimmer, because the human eye cannot recognise the LED being turned on and off very fast. Now connect the piezo buzzer instead of the LED, see Figure 7. You should hear a 300 Hz (not very clear) tone.

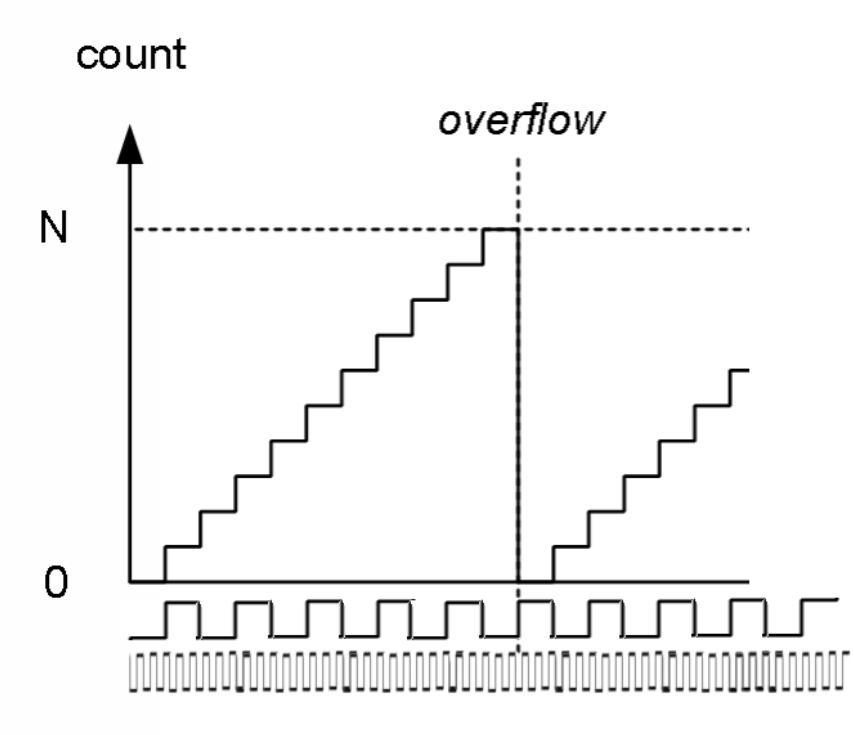


Figure 4: Counter overflow

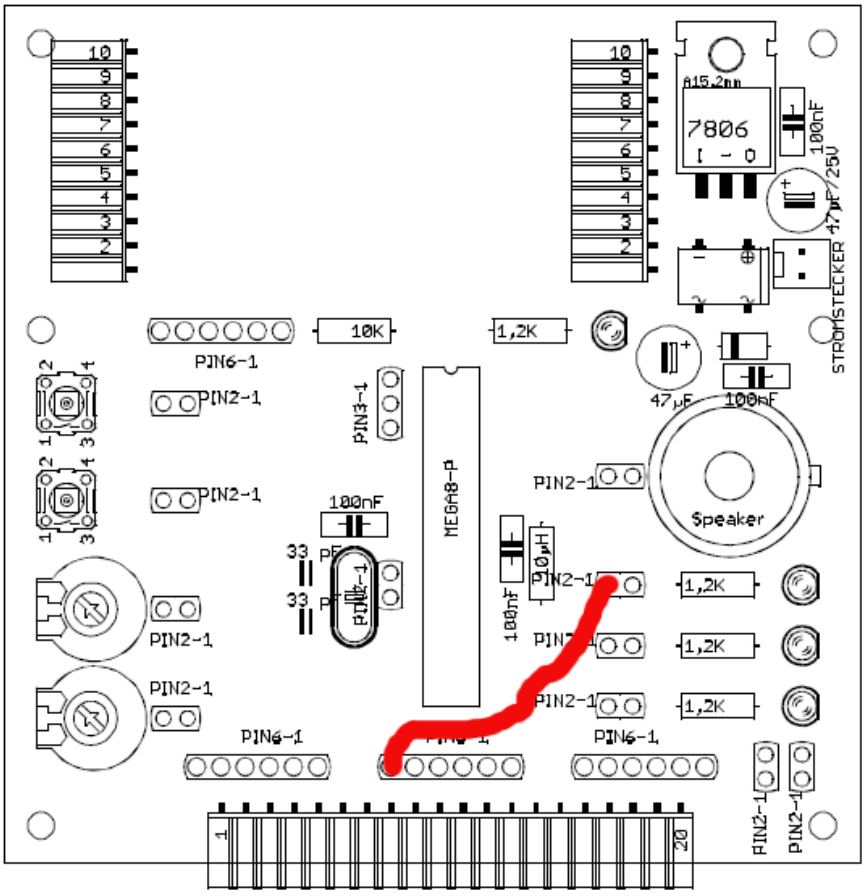


Figure 5: LED connection on the board

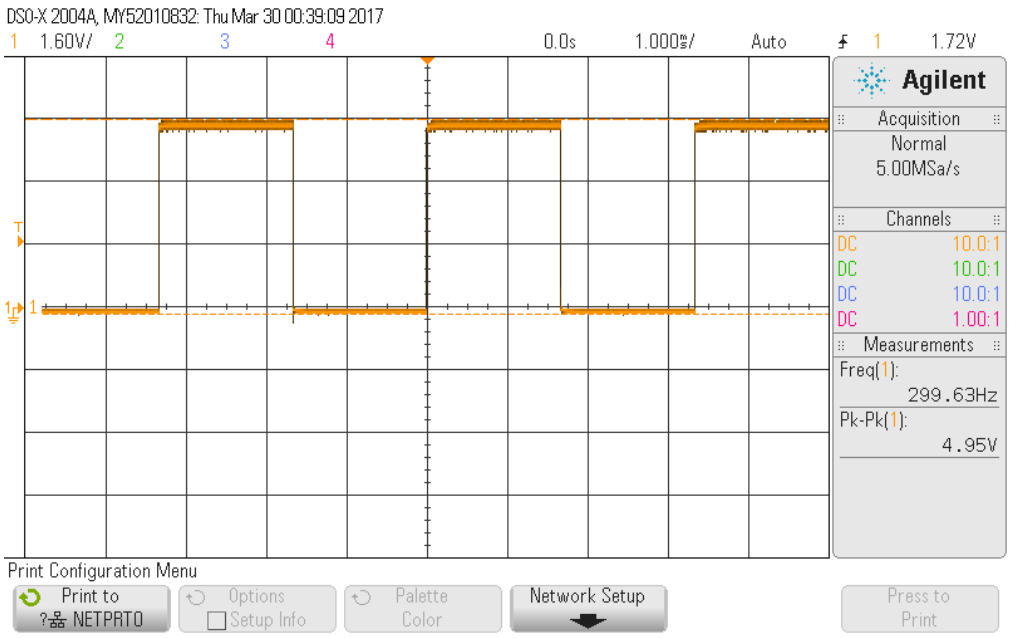


Figure 6: PWM signal 300 Hz captured with oscilloscope

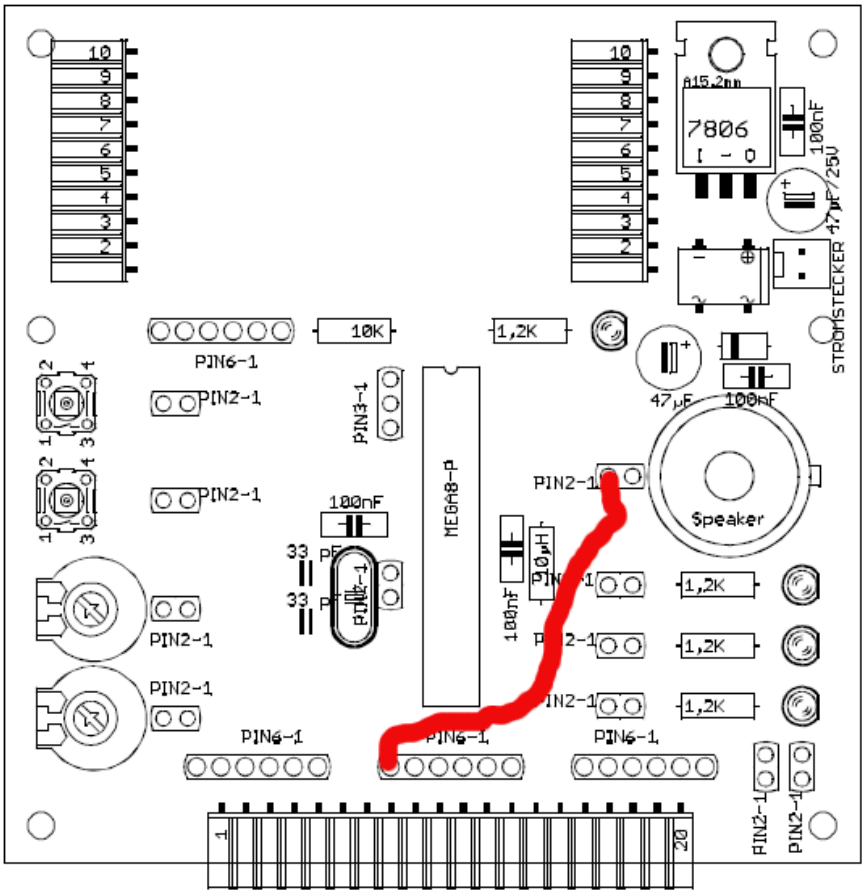


Figure 7: Audio output connection on the board

If all at this point is complete and works fine, inform the staff. You must be able to show and explain all steps you have done. The next steps are optional.

3 Using two Timer/Counter

This section is optional.

Use two Timer/Counter to play a song. One timer is to generate the tone pitch, and the other for the tone length.

The tones for ”silent night” are:

Tone	**Text**	**Duration(sec)**
G	Silent	0.34375
A		0.09375
G		0.21875
E	Night	0.71875
G	Holy	0.34375
A		0.09375
G		0.21875
E	Night	0.71875
G1	All	0.46875
G1	Is	0.21875
B	Calm	0.71875
C	All	0.46875
C	Is	0.21875
G	Bright	0.71875

Frequencies:

- A = 440 Hz
- B = 492,88 Hz
- C = 523,25 Hz
- E = 329,63 Hz
- G = 392 Hz
- G1 = 783,99 Hz

Here is a hint to make eventual changes more easy:

```
#define A 36
#define B 32
#define C 30
```

```
#define E 47
#define G 40
#define G1 20
#define END 1


uint8_t uchNote[] = {G, A, G, E, G, A, G, E, G1, G1, B, C, C, G, END}
```

HOCHSCHULE
RHEIN-WAAL
Rhine-Waal University
of Applied Sciences

LABORATORY

Microcontroller

4

EXPERIMENT:

Interrupts

Please read the whole document before starting the experiment. This document contains 7 preparation jobs. It is mandatory to make all preparation jobs in written form! Without a written document that includes all preparation jobs it is not possible to pass this experiment.

1 Interrupts Basics

The simple way for a running programs is to execute it step after step. For some tasks a very fast reaction is required. To react on defined events the program need to ”look” for the event as often as possible. This can be a problem, it is inefficient and bad program code, see Figure 1.

To explain this, think about an example from the real world. A person is cleaning up his home and waiting for a guest. It is an annoying way to interrupt cleaning every minute and check if the guest is waiting in front of the door. The real world solution for this problem is the doorbell. The guest will use the doorbell, in the microcontroller word this is called interrupt.

A microcontroller interrupt can have several sources. In the real world this could be the doorbell, the phone bell or an alarm clock. Most microcontrollers have a sleep mode, it is used to save energy. Interrupts are often the only way to wake up a microcontroller.

Typical interrupt events:

- external interrupts (digital input)
- Timer / Counter compare register
- Timer / Counter overflow
- communication events (incoming data, sending complete, ...)
- watchdog

Prepare 1: What is a watchdog in a microcontroller?

On most microcontrollers every event has his own event handler. This handler is called **I**nterrupt **S**ervice **R**outine (ISR). The interrupt hardware stops the normal program and starts the ISR. After the ISR is done the normal program will continue, see Figure 2.

To prevent troubleshooting between the normal program and the ISR, the ISR must be as short as possible. The remaining time for the normal program must be long enough.

The normal program can disable interrupts. This is necessary if:

- normal program and ISR share memory (prevent double access)
- a part of the normal program is time critical

Prepare 2: Is it a good idea to activate the sleep mode with disabled interrupts?

The CPU can work with enabled or disabled interrupts. By default the program can not be interrupted. The program must enable the global interrupt switch with *sei();*

Inside an ISR other interrupt events are only stored and processed back to back. While an ISR is running the global interrupt switch is disabled until the ISR is done. This is done automatically.

It is possible to allow interrupts inside an ISR. This generates a low priority ISR. A program can become very complex with interruptable ISR, use it only if it is really necessary.

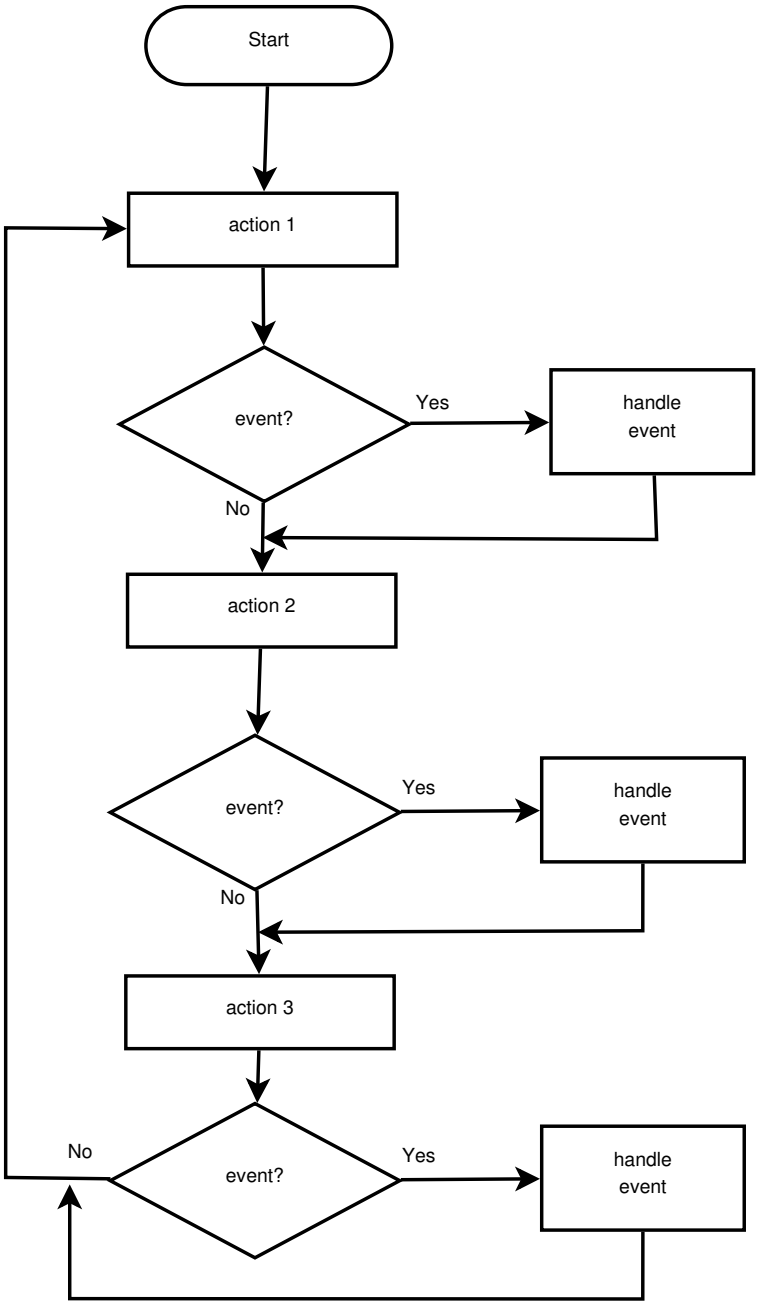


Figure 1: Without Interrupts

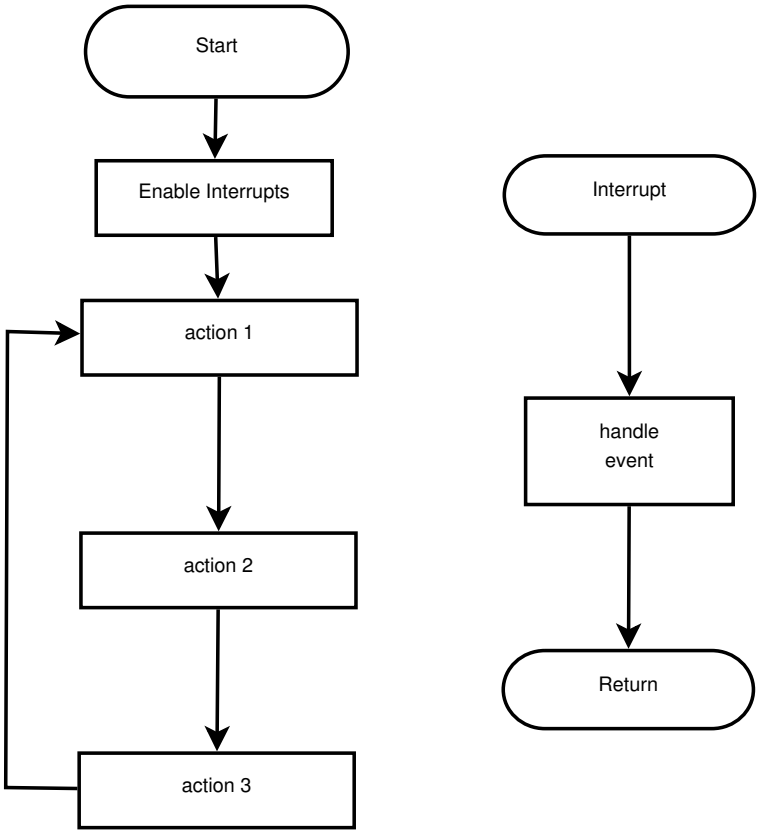


Figure 2: With Interrupts

Bit	7	6	5	4	3	2	1	0	
0x1D (0x3D)	–	–	–	–	–	–	INT1	INT0	EIMSK
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 3: External Interrupt Mask Register

1.1 Interrupts on the Atmel ATmega88PA

The interrupt handling is controlled with registers. Inside the Status Register is the bit: **Global Interrupt Enable**. In C there is a program library with functions to set this bit: <avr/interrupt.h>

To enable interrupts use: void sei(void);
To disable interrupts use: void cli(void);

This is the general interrupt setting. The interrupt sources must be enabled separately! It is like a master switch and device switches.

Please note that the hardware executes interrupts without any security check. If an interrupt is enabled but not defined the microcontroller can crash (no hardware damage)!

The input pins PD2 and PD3 can be used for external inputs. To enable the external interrupts set the bits INT0 / INT1 in the EIMSK register, see Figure 3.

Most of the physical pins have multiple functions. On the MyAVR board the pins for external interrupts are used to communicate with the LCD display. To use this interrupts, please disconnect the LCD display and do not add any LCD functions to the source code.

1.2 ISR in C

In C every Interrupt is a simple function. Because the ISR is called by an event and not by another software function there is no return value. To distinguish between the different interrupt events, there is a event parameter. Look at the defines in the following list. The list is an excerpt and not complete.

```
//...
/* Id : iom8.h, v1.132005/10/3022 : 11 : 23joerg_wunschExp */

/* avr/iom88p.h - definitions for ATmega88pa */
//...

/* Interrupt vectors */

/* External Interrupt Request 0 */
#define INT0_vect _VECTOR(1)
#define SIG_INTERRUPT0 _VECTOR(1)

/* External Interrupt Request 1 */
#define INT1_vect _VECTOR(2)
#define SIG_INTERRUPT1 _VECTOR(2)

/* Timer/Counter2 Compare Match */
#define TIMER2_COMP_vect _VECTOR(3)
#define SIG_OUTPUT_COMPARE2 _VECTOR(3)

/* Timer/Counter2 Overflow */
#define TIMER2_OVF_vect _VECTOR(4)
#define SIG_OVERFLOW2 _VECTOR(4)

/* Timer/Counter1 Capture Event */
#define TIMER1_CAPT_vect _VECTOR(5)
#define SIG_INPUT_CAPTURE1 _VECTOR(5)

/* Timer/Counter1 Compare Match A */
#define TIMER1_COMPA_vect _VECTOR(6)
#define SIG_OUTPUT_COMPARE1A _VECTOR(6)

/* Timer/Counter1 Compare Match B */
#define TIMER1_COMPB_vect _VECTOR(7)
```

```
#define SIG_OUTPUT_COMPARE1B _VECTOR(7)

//...
```

Examples for some ISR functions in C:

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

//...

void main()
{
//...
sei();
//...
}

ISR(INT0_vect)
{
/* Interrupt Code */
}

ISR(TIMER0_OVF_vect)
{
/* Interrupt Code */
}

ISR(USART_RXC_vect)
{
/* Interrupt Code */
}
```

Please note, the compiler will generate different ISR(parameter) functions. The parameter is not the same like a normal C function with a parameter, in this case it is a macro.

2 Using interrupts

Before flashing and running the programs remove the LCD display from the board and connect:

- Pin D2 to key 1
- Pin D3 to key 2

- Pin B0 to red LED
- Pin B1 to yellow LED
- Pin B2 to green LED
- Pin C0 to potentiometer 1

Use the template in "Template/Experiment4", the init function in this template is not complete.

2.1 Naive Projection

Write a program (without using interrupts) that:

- enables the red LED when button 1 is pressed
- disables the red LED when button 2 is pressed

If this program runs, expand the program:

- the yellow LED should blink with 0.5 sec delay (use _delay_ms(250) two times)

  - In order to use _delay_ms(250) it is necessary to add #define F_CPU 8000000UL and #include <util/delay.h> to the code.

Now you can see, that your program has a bad reaction on the buttons. This is because the device is hanging while the delay function is waiting.

2.2 Digital Input Interrupts

To solve the problem from 2.1 it is time to interrupt the CPU in the waiting time. Move the enabling and disabling from the main function to the interrupts INT0 and INT1.

Prepare 3: What is the correct setting from the EICRA register (datasheet p. 71) to react on the falling edges for INT0 and INT1 (look up in the data sheet)?

The first step is to set the correct register values. To use INT0 and INT1 it is required to:

- set bits in register EICRA
- set bits in register EIMSK
- enable the global interrupt handling (already done here)
- write the two ISR functions

2.3 Timer Interrupts

Prepare 4: How do you set TCCR1B (datasheet p. 133) to set the prescaler to 256? What is the time between the overflows (8 MHz, 16 Bit)?

Prepare 5: What is the correct setting from the TIMSK1 register (datasheet pp. 135-136) to enable the overflow interrupt for Timer/Counter 1?

Use Timer/Counter1 to flash the green LED with the overflow interrupt from this timer.

If all at this point is complete and works fine, inform the staff. You must be able to show and explain all steps you have done. The next steps are optional.

2.4 Analog input recap

This section is optional.

Prepare 6: How to set the ADMUX and ADCSRA register? Use Avcc as reference voltage, PC0/ADC0 as input and set the prescaler to 64.

Expand the main loop. Read the value from the potentiometer one (ADC0) and set the prescaler from Timer/Counter to one of following values:

- 1
- 8
- 64
- 256

Prepare 7: How to set the TCCR1B for the four speed steps?

The potentiometer should work as a four step speed regulator for the flashing green LED.

HOCHSCHULE
RHEIN-WAAL
Rhine-Waal University
of Applied Sciences

LABORATORY

Microcontroller

5

EXPERIMENT:

Seven Segment Counter

Please read the whole document before starting the experiment. This document contains 5 preparation jobs. It is mandatory to make all preparation jobs in written form! Without a written document that includes all preparation jobs it is not possible to pass this experiment.

1 Seven Segment Display

A Seven-segment displays is a cheap method to display numbers 0-9 using only 7 LEDs (or light bulbs). The segments are named A to G and DP for “decimal point”. See Figure 1.

For displaying a zero the segments must to be set to:

A=1, B=1, C=1, D=1, E=1, F=1, G=0

Prepare 1: Create a table for all numbers from 0 to 9.

1.1 The "breadboard"

Today you will use a so called ”breadboard” for plugging the circuits, see Figure 2.

A breadboard is often used for prototyping small circuits. The bread board is broken up into positive (+) and negative (-) ”power rails” which span the height of the board. The positive rails are colored in Red, while the negative rails are colored in Blue. These rails are internally connected vertically. In the middle of the bread board are multiple groups of holes organized in vertical columns and numbered every 5 (for reference). Each group consists of 5 pin sockets wired together horizontally. These groups are not connected over the middle horizontal break of the board, nor do they connect vertically.

Prepare 2: Become familiar with the breadboards internal connections!

1.2 Kingbright Sx39-11

The display, we are going to use is a Kingbright Sx39. A snippet from the datasheet is in Figure 3 and Figure 4.

Please note: We have the SA39 and the SC39 in the lab. Check what you get, this two devices are different!
Prepare 3: Make sure you understand how seven-segment displays work and where to connect signals and power supply on the SA39-11/SC39-11.

Read the datasheet! We want to operate the LEDs at a maximum of 6 mA. Since you attended the analog electronics lab, you should know how to limit the current through an LED. Calculate with an LED forward voltage from 2V. The datasheet is appended to this document at page 5.

Prepare 4: Calculate the necessary resistors (Supply voltage will be 5 V).

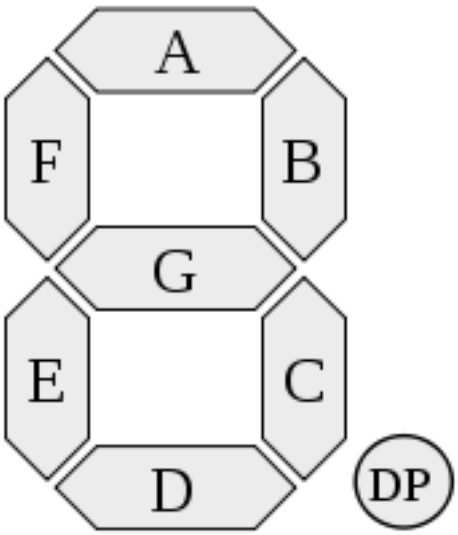


Figure 1: Seven Segment Display

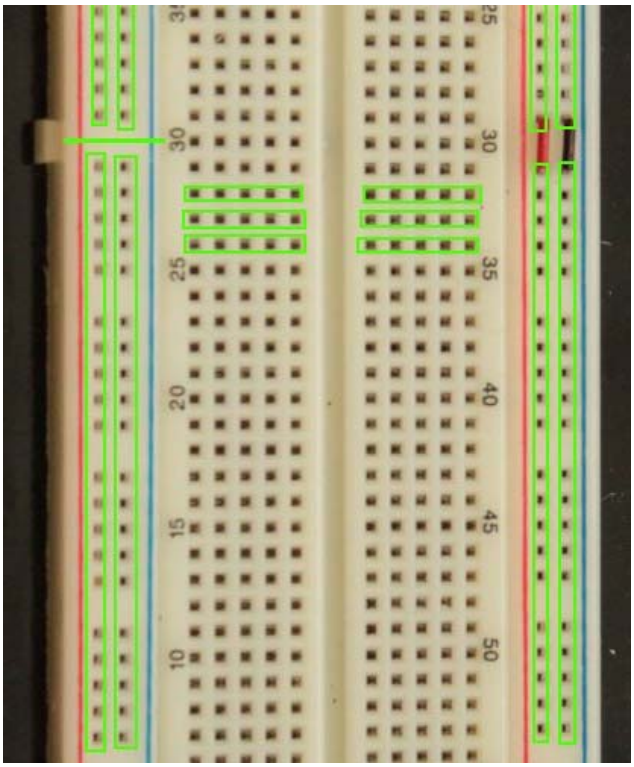


Figure 2: "Breadboard"

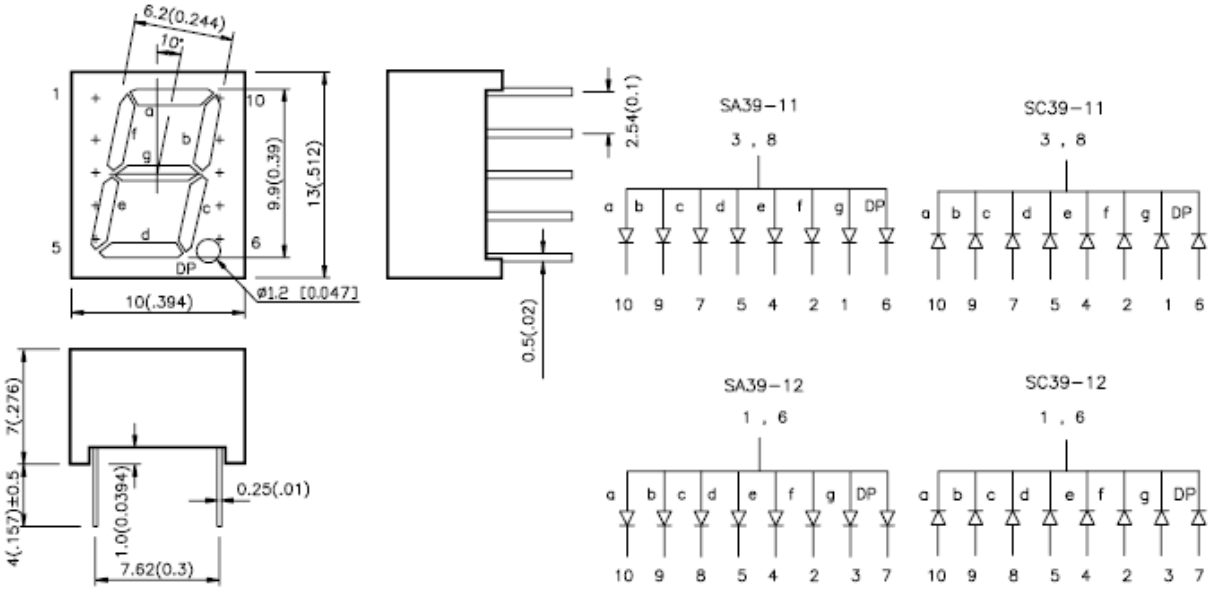


Figure 3: SC39-11

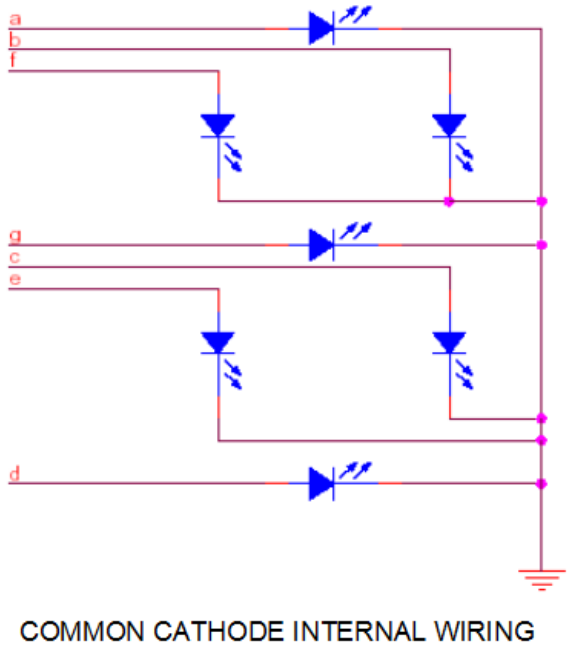


Figure 4: SC39-11

2 Seven Segment Counter

What we want at the end of this task is a working “counter”, see Figure 5.

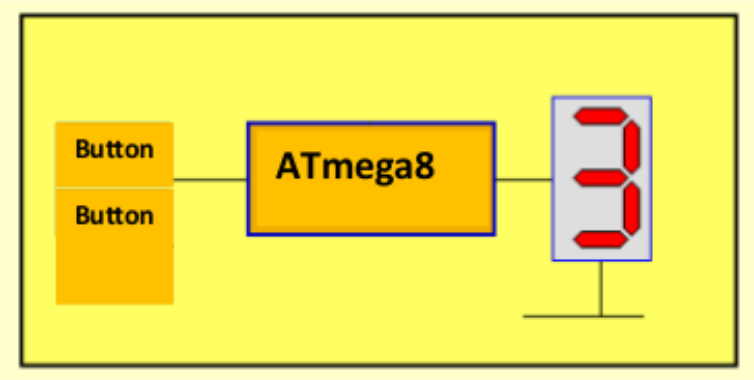


Figure 5: Counter with seven segment display

There should be a button as input and a seven segment display showing the number of switch operations. This looks very simple, but you will get a few typical problems like:

- contact bouncing
- signal edge evaluation
- bit manipulations
- counting / limits

The display will be mounted on a breadboard. You can find a short explanation of a breadboard in 1.1. Please try to become familiar with this device. If the provided information is not sufficient, use additional material from the Internet.

Prepare 5: Design a general idea, how to count the input pulses of the button. We have been using the buttons as a steady input (either on or off) already in the previous labs, so you basically know how to read inputs. Start with flow charts and later on, turn the charts into a C code.

If the counter with one button works fine, add a second button for counting up and down. If both buttons are pressed the counter should reset to zero.

There is a template, read this program first, understand and complete it.

You have to write/complete a function to set the outputs. Please use for the input:

- **Key 1 (up):** PD2
- **Key 2 (down):** PD3

and for the output (fill out last column):

	MyAVR	Sx39-11 Pin
A	PC2	
B	PC1	
C	PC0	
D	PB0	
E	PB1	
F	PB2	
G	PB3	
	GND	

If all at this point is complete and works fine, inform the staff. You must be able to show and explain all steps you have done. The next steps are optional.

Use one analog to digital converter to create a simple voltmeter. Measure the voltage from a potentiometer. The output over the seven segment can only show one digit, so only display a rounded value without decimal places.

3 Appendix

Datasheet: Kingbright® Sx39

Kingbright®

9.9mm (0.39INCH) SINGLE DIGIT NUMERIC DISPLAYS

SA39-11 SC39-11

SA39-12 SC39-12

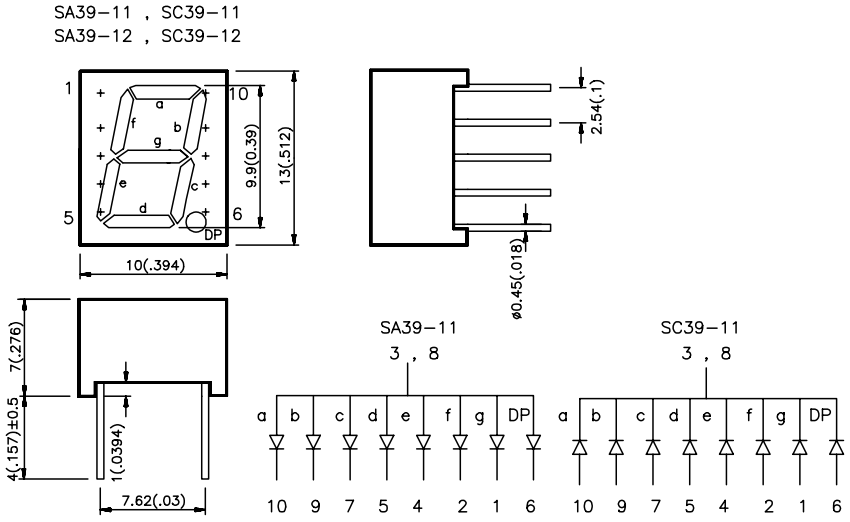
Features

- ●0.39 INCH DIGIT HEIGHT.
- ●LOW CURRENT OPERATION.
- ●EXCELLENT CHARACTER APPEARANCE.
- ●EASY MOUNTING ON P.C. BOARDS OR SOCKETS.
- ●I.C. COMPATIBLE.
- ●CATEGORIZED FOR LUMINOUS INTENSITY,
- ●YELLOW AND GREEN CATEGORIZED FOR COLOR.
- ●MECHANICALLY RUGGED.
- ●STANDARD : GRAY FACE, WHITE SEGMENT.

Description

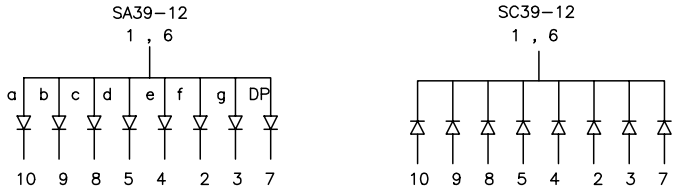
The Bright Red source color devices are made with Gallium Phosphide Red Light Emitting Diode.
The Green source color devices are made with Gallium Phosphide Green Light Emitting Diode.
The High Efficiency Red source color devices are made with Gallium Arsenide Phosphide on Gallium Phosphide Orange Light Emitting Diode.
The Yellow source color devices are made with Gallium Arsenide Phosphide on Gallium Phosphide Yellow Light Emitting Diode.
The Super Bright Red source color devices are made with Gallium Aluminum Arsenide Red Light Emitting Diode.

Package Dimensions & Internal Circuit Diagram



Notes:
1. All dimensions are in millimeters (inches), Tolerance is ±0.25(0.01")unless otherwise noted.
2. Specifications are subjected to change whitout notice.

Internal Circuit Diagram



Selection Guide

Part No.	Dice	Iv (ucd) @ 10 mA		Description
		Min.	Max.	
SA39-11HWA SA39-12HWA	BRIGHT RED (GaP)	560	2200	Common Anode
SC39-11HWA SC39-12HWA				Common Cathode
SA39-11EWA SA39-12EWA	HIGH EFFICIENCY RED (GaAsP/GaP)	2200	9000	Common Anode
SC39-11EWA SC39-12EWA				Common Cathode
SA39-11GWA SA39-12GWA	GREEN (GaP)	1400	5600	Common Anode
SC39-11GWA SC39-12GWA				Common Cathode
SA39-11YWA SA39-12YWA	YELLOW (GaAsP/GaP)	1400	5600	Common Anode
SC39-11YWA SC39-12YWA				Common Cathode
SA39-11SRWA SA39-12SRWA	SUPER BRIGHT RED (GaAlAs)	9000	31000	Common Anode
SC39-11SRWA SC39-12SRWA				Common Cathode

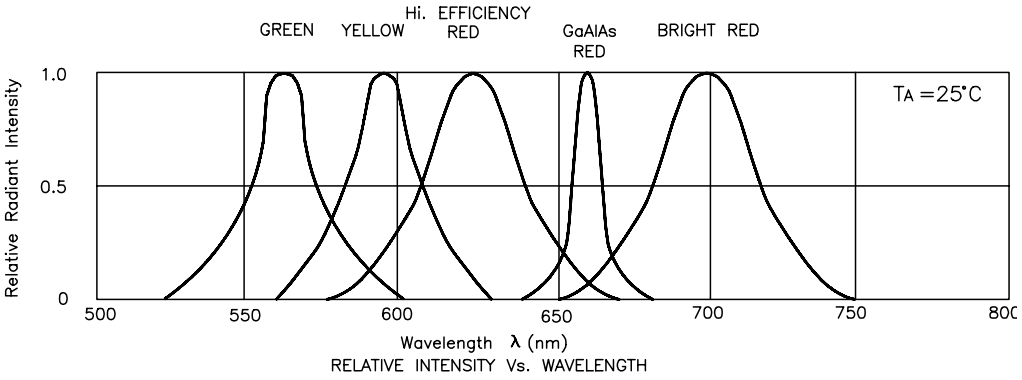
Electrical / Optical Characteristics at T_A=25°C

Symbol	Parameter	Device	Typ.	Max.	Units	Test Conditions
λ_{peak}	Peak Wavelength	Bright Red High Efficiency Red Green Yellow Super Bright Red	700 625 565 590 660		nm	IF=20mA
$\Delta\lambda_{1/2}$	Spectral Line Halfwidth	Bright Red High Efficiency Red Green Yellow Super Bright Red	45 45 30 35 20		nm	IF=20mA
C	Capacitance	Bright Red High Efficiency Red Green Yellow Super Bright Red	40 12 45 10 95		pF	VF=0V;f=1MHz
V_F	Forward Voltage	Bright Red High Efficiency Red Green Yellow Super Bright Red	2.0 2.0 2.2 2.1 1.85	2.5 2.5 2.5 2.5 2.5	V	IF=20mA
I_R	Reverse Current	All	10		uA	VR = 5V

Absolute Maximum Ratings at T_A=25°C

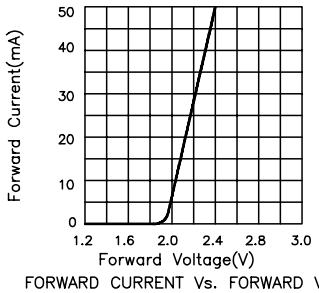
Parameter	Bright Red	High Efficiency Red	Green	Yellow	Super Bright Red	Units
Power dissipation	120	105	105	105	100	mW
DC Forward Current	25	30	25	30	30	mA
Peak Forward Current [1]	150	150	150	150	150	mA
Reverse Voltage	5	5	5	5	5	V
Operating/Storage Temperature	-40° C To +85 ° C					
Lead Soldering Temperature [2]	260° C For 5 Seconds					

Notes:
1.1/10 Duty Cycle, 0.1ms Pulse Width.
2. 4mm below package base.

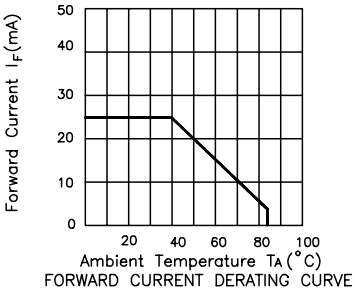


RELATIVE INTENSITY Vs. WAVELENGTH

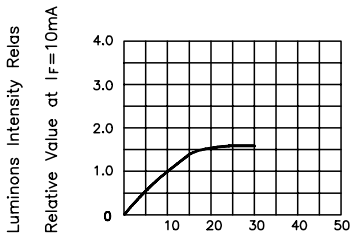
Bright Red



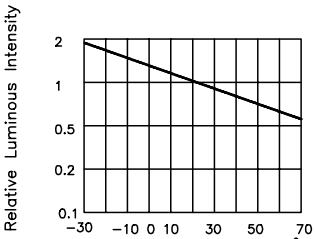
FORWARD CURRENT Vs. FORWARD VOLTAGE



FORWARD CURRENT DERATING CURVE

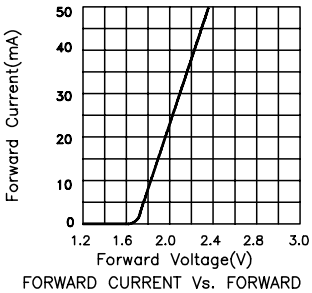


LUMINOUS INTENSITY Vs. FORWARD CURRENT

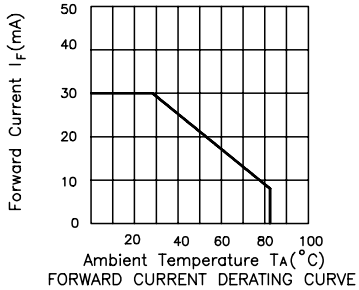


LUMINOUS INTENSITY Vs. AMBIENT TEMPERATURE

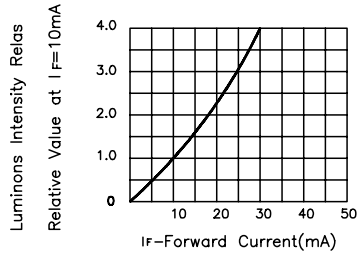
High Efficiency Red



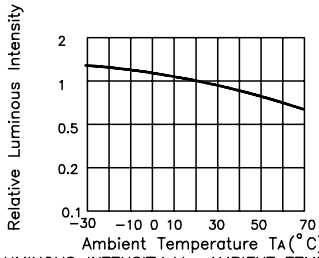
FORWARD CURRENT Vs. FORWARD VOLTAGE



FORWARD CURRENT DERATING CURVE

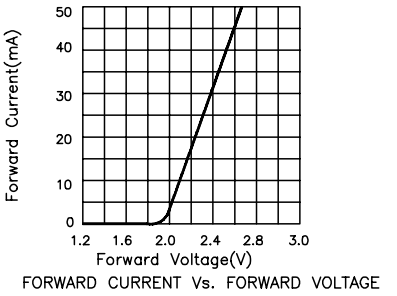


LUMINOUS INTENSITY Vs. FORWARD CURRENT

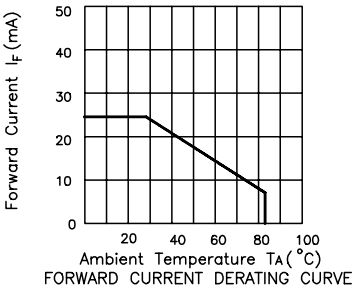


LUMINOUS INTENSITY Vs. AMBIENT TEMPERATURE

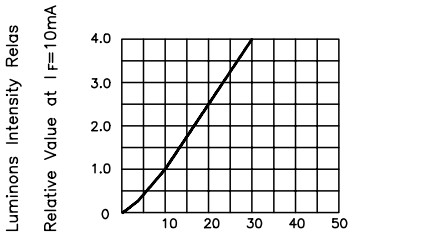
Green



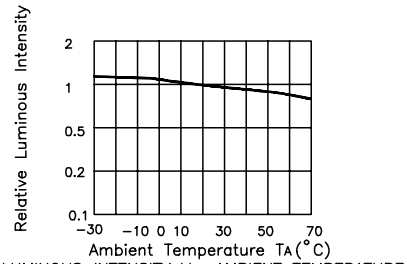
FORWARD CURRENT Vs. FORWARD VOLTAGE



FORWARD CURRENT DERATING CURVE

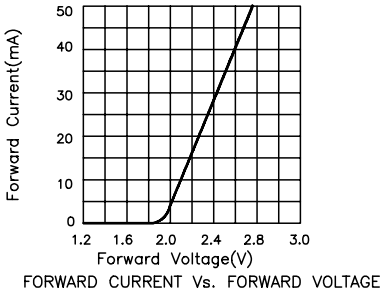


LUMINOUS INTENSITY Vs. FORWARD CURRENT

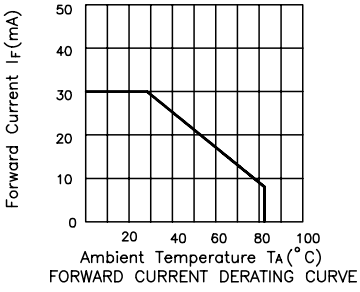


LUMINOUS INTENSITY Vs. AMBIENT TEMPERATURE

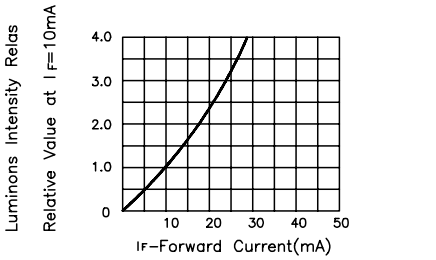
Yellow



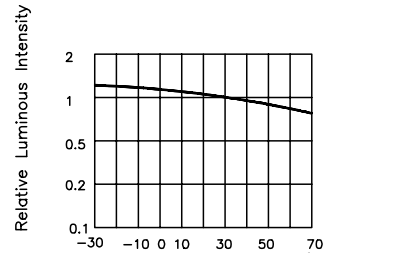
FORWARD CURRENT Vs. FORWARD VOLTAGE



FORWARD CURRENT DERATING CURVE

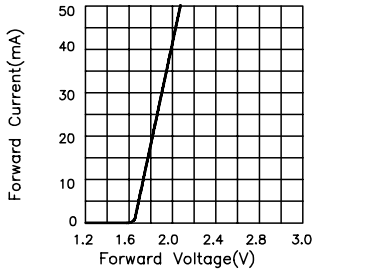


LUMINOUS INTENSITY Vs. FORWARD CURRENT

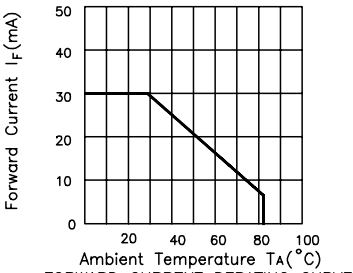


LUMINOUS INTENSITY Vs. AMBIENT TEMPERATURE

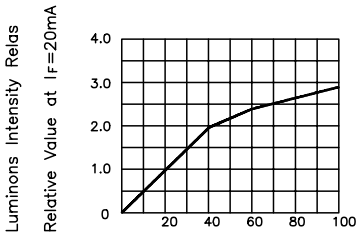
Super Bright Red



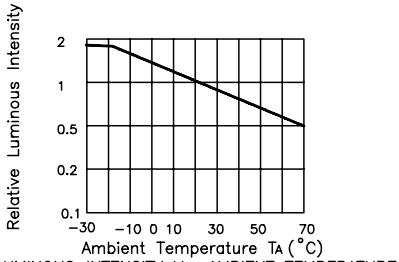
FORWARD CURRENT Vs. FORWARD VOLTAGE



FORWARD CURRENT DERATING CURVE



LUMINOUS INTENSITY Vs. FORWARD CURRENT



LUMINOUS INTENSITY Vs. AMBIENT TEMPERATURE

HOCHSCHULE
RHEIN-WAAL
Rhine-Waal University
of Applied Sciences

LABORATORY

Microcontroller

6

EXPERIMENT:

I^2C Bus 1: EEPROM

Please read the whole document before starting the experiment. This document contains 7 preparation jobs. It is mandatory to make all preparation jobs in written form! Without a written document that includes all preparation jobs it is not possible to pass this experiment.

1 The I²C Bus

The I²C Bus (Inter-Integrated Circuit) was developed in 1982 to realize the communication between integrated circuits. It uses serial data transfer.

A microcontroller has normally memory, input/output, timer, and so on. Sometimes it is necessary to add more memory, input/output, ... to a microcontroller. This is the typical usage for the I²C Bus:

- **A short distance:** No error check or error correction
- **The devices are on the same board:** It uses only two wires to make the layout design more simple
- **"Active parts" like microcontroller and "passive" parts like memory:** Master/Slave structure

A typical usage is one master and multiple slave devices. Multiple master devices are possible, but we will not use this inside the lab.

1.1 Electrical part

Only two signal wires are used for the I²C Bus:

- **SDA:** For the data
- **SCL:** For the clock signal

Because the I²C Bus is designed for the communication inside one circuit and not for a data transfer to different machines, a global ground is a natural consequence for the I²C Bus.

The master generates the clock signal and has the control over the whole bus. A slave only sends data when the master requests this. To see how the devices are connected, see Figure 1.

The I²C Bus has a maximum speed of:

- **Standard Mode:** 100 kBit/s
- **Fast Mode:** 400 kBit/s
- **Fast Mode Plus:** 1 MBit/s

The speed is controlled by the master. This devices generates the clock signal, that is global for the whole bus. So the devices do not need a precise clock generator for the serial communication.

In Figure 1 you can see there are pullup resistors for both signals. The bus subscribers have open collector connections to the two bus signals, see Figure 2.

Prepare 1: Is a short circuit possible, when a fault happens and more than one device try to write data to the bus?

Prepare 2: Is a broken device able to bear down the communication on the whole bus?

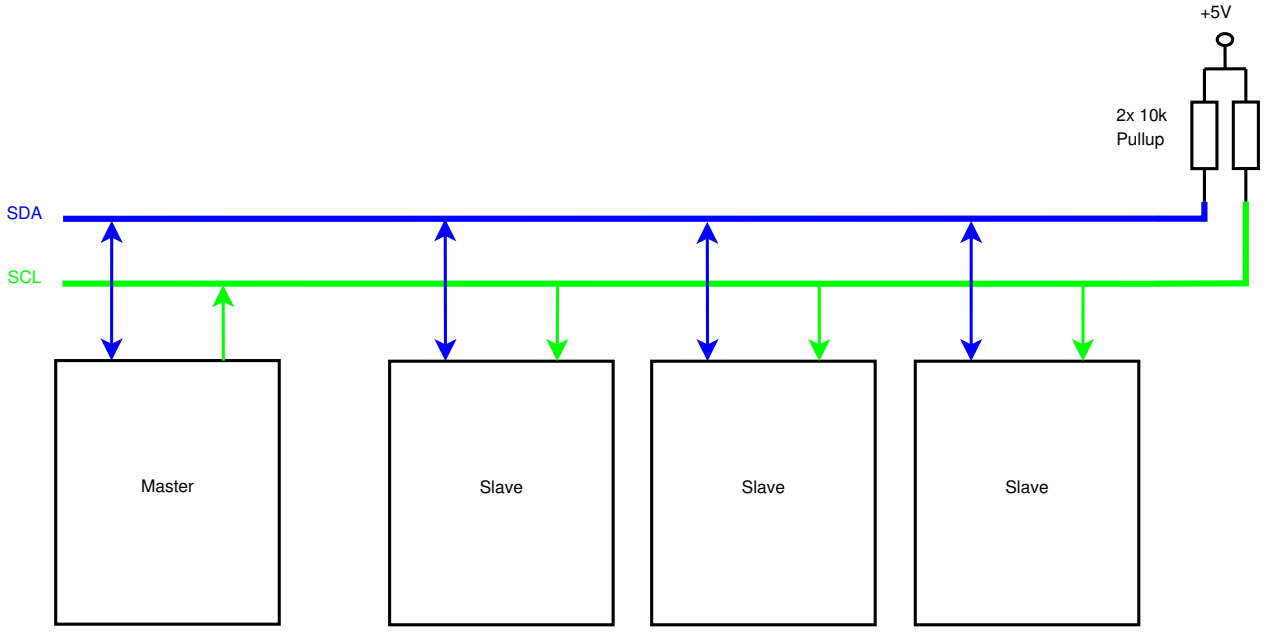


Figure 1: Typical I^{2}C Bus circuit

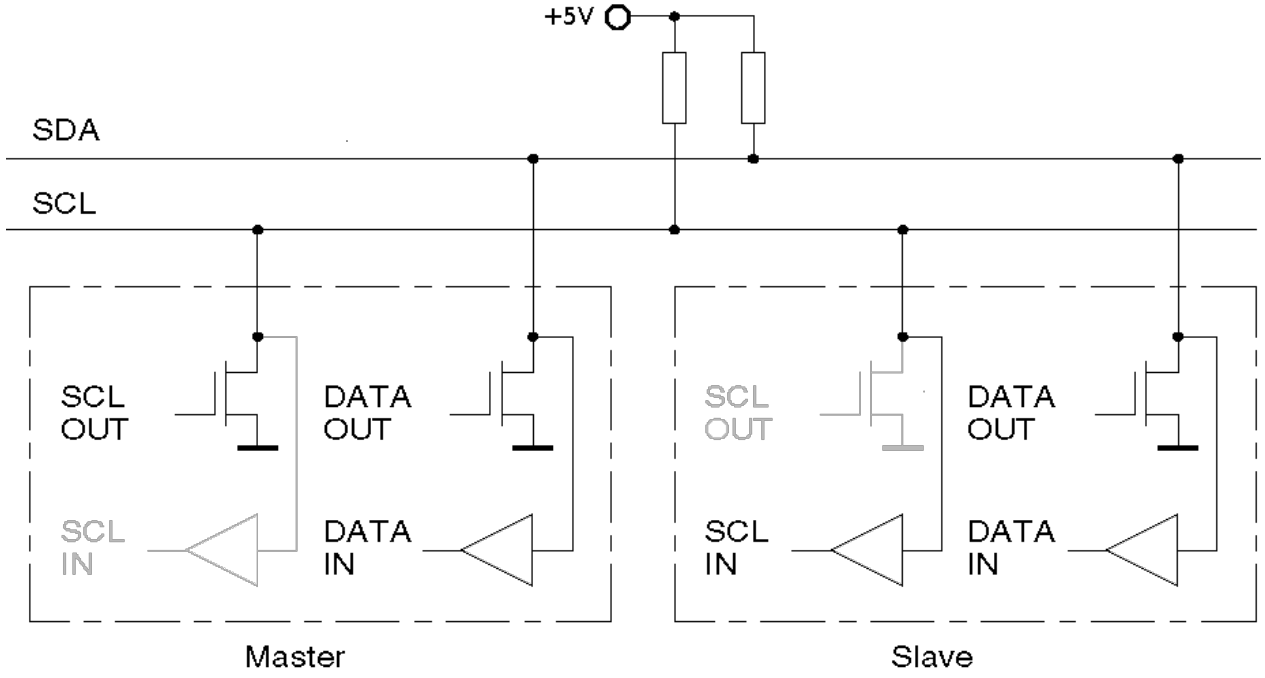


Figure 2: I^{2}C Bus driver

1.2 I²C Bus on the Atmel ATmega88PA

In the lab we use a microcontroller from Atmel. Atmel devices use the name TWI (Two Wire Interface) for ı2cbus . There is only a licence problem with the name ı2cbus , the technical part is exactly the same. Some newer Atmel datasheets use the name 2-wire Serial Interface.

Prepare 3: What are the page numbers in the datasheet for TWI?

Prepare 4: List all registers (only the names, not every bit) to control the TWI hardware.

To use the TWI there is a pre written library in the template:

- i2c_master.c
- i2c_master.h

The .c file is incomplete, you have to complete it.

In the lab we work with memory like devices. Every device has an internal position pointer. To write or read data, it is important to read from the right position inside the memory, from the right device.

Prepare 5: How many slave devices are possible (theoretical)? What is the difference between read and write address?

When the I²C Bus is opened, the first byte from the master is the device address. The device that has this address activates, all other devices will ignore the following bus traffic. The second step is to set the position of the pointer inside the memory device. From now it is possible to read or write data. The pointer increases automatically.

To read data with this library via I²C Bus:

1. i2c_master_open_write(adr); //Device address, open to write
2. i2c_master_write(pos); //Set the position pointer of the device
3. i2c_master_open_read(adr); //Re-Open the same device to read
4. x = i2c_master_read_next(); //Read a byte
5. y = i2c_master_read_next(); //Read a byte
6. z = i2c_master_read_last(); //Read last byte
7. i2c_master_close(); //Close the device access

To write data with this library via I²C Bus:

1. i2c_master_open_write(adr); //Device address, open to write
2. i2c_master_write(pos); //Set the position pointer of the device
3. i2c_master_write(x); //Write a byte
4. i2c_master_write(y); //Write a byte
5. i2c_master_close(); //Close the device access

2 EEPROM

EEPROM (Electrically Erasable Programmable Read-Only Memory) is a memory, that can save data without any power source. This is a good memory to save a system configuration. For example a hand calculator can store setting on an EEPROM. So if you change the battery, it is not necessary to enter all settings again.

In the Lab we will use an ST 24C02. This integrated circuit can have different addresses depending of the state of three input pins. The hardware inside the lab has jumper to select the value for this pins.

Prepare 6: Read the datasheet of the 24C02 and write down all possible addresses in Hex (with the last bit zero).

2.1 Task EEPROM

Connect the LCD display and the myTWI EEPROM board and complete the template. Use the wires to connect:

- Key1 to PB0
- Key2 to PB1
- Poti1 to PC0

Read the template and understand the code before you edit it. The template has different .c files. Read all files, except the LCD files. Hint: You can use the search function of the editor and search for "TODO". You need to use datasheets of the microcontroller and the EEPROM to solve this lab.

Your program should do the following:

- Show the up to date potentiometer value on the LCD
- The buttons are to save and load the value to/from the EEPROM
- Handle the potentiometer value as an 16 bit value

To test that your program works good, do the following steps:

- Set any value with the potentiometer
- Press key 1 (save)
- Press key 2 (load)
- Check that the value is now shown on the LCD in the second line
- Disconnect the USB wire, wait a few seconds, reconnect the USB wire
- Press key 2 (load) and check the value

If all at this point is complete and works fine, inform the staff. You must be able to show and explain all steps you have done. The next steps are optional.

Expand the program in a way that it is possible to store and load more than one value. Make this possible with only the two buttons and the one potentiometer.

Prepare 7: How many 16 bit values can be stored inside the used EEPROM?

3 Appendix

Datasheet: ST24C02 EEPROM

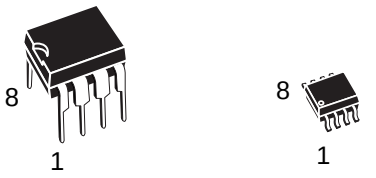
SGS-THOMSON MICROELECTRONICS

ST24/25C02, ST24C02R ST24/25W02

SERIAL 2K (256 x 8) EEPROM

NOT FOR NEW DESIGN

- 1 MILLION ERASE/WRITE CYCLES with 40 YEARS DATA RETENTION
- SINGLE SUPPLY VOLTAGE:
  - 3V to 5.5V for ST24x02 versions
  - 2.5V to 5.5V for ST25x02 versions
  - 1.8V to 5.5V for ST24C02R version only
- HARDWARE WRITE CONTROL VERSIONS: ST24W02 and ST25W02
- TWO WIRE SERIAL INTERFACE, FULLY I^2C BUS COMPATIBLE
- BYTE and MULTIBYTE WRITE (up to 4 BYTES)
- PAGE WRITE (up to 8 BYTES)
- BYTE, RANDOM and SEQUENTIAL READ MODES
- SELF TIMED PROGRAMMING CYCLE
- AUTOMATIC ADDRESS INCREMENTING
- ENHANCED ESD/LATCH-UP PERFORMANCES
- ***ST24C/W02 are replaced by the M24C02***
- ***ST25C/W02 are replaced by the M24C02-W***
- ***ST24C02R is replaced by the M24C02-R***



PSDIP8 (B)
0.25mm Frame

SO8 (M)
150mil Width

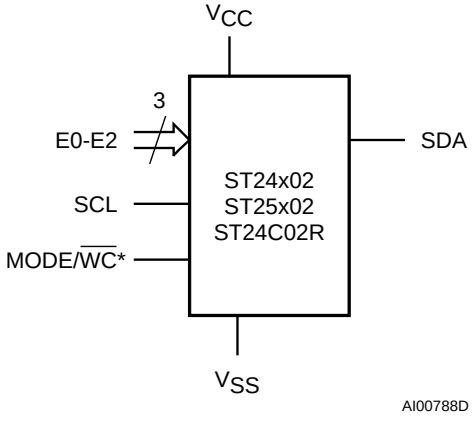
DESCRIPTION

This specification covers a range of 2K bits I^2C bus EEPROM products, the ST24/25C02, the ST24C02R and ST24/25W02. In the text, products are referred to as ST24/25x02, where "x" is: "C" for Standard version and "W" for hardware Write Control version.

Table 1. Signal Names

E0-E2	Chip Enable Inputs
SDA	Serial Data Address Input/Output
SCL	Serial Clock
MODE	Multibyte/Page Write Mode (C version)
$\overline{WC}$	Write Control (W version)
V_{CC}	Supply Voltage
V_{SS}	Ground

Figure 1. Logic Diagram



Note: $\overline{WC}$ signal is only available for ST24/25W02 products.

This is information on a product still in production but not recommended for new design

Figure 2A. DIP Pin Connections

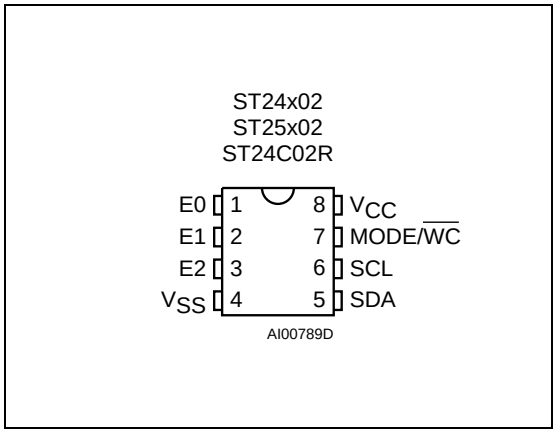


Figure 2B. SO Pin Connections

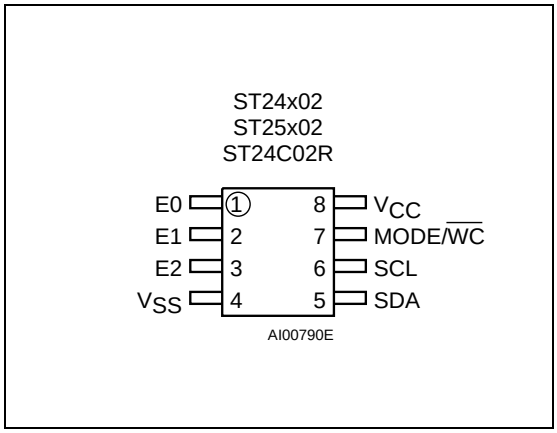


Table 2. Absolute Maximum Ratings (1)

Symbol	Parameter		Value	Unit
T_A	Ambient Operating Temperature		–40 to 125	°C
T_{STG}	Storage Temperature		–65 to 150	°C
T_{LEAD}	Lead Temperature, Soldering	(SO8 package) 40 sec (PSDIP8 package) 10 sec	215 260	°C
V_{IO}	Input or Output Voltages		–0.6 to 6.5	V
V_{CC}	Supply Voltage		–0.3 to 6.5	V
V_{ESD}	Electrostatic Discharge Voltage (Human Body model) (2)		4000	V
	Electrostatic Discharge Voltage (Machine model) (3)		500	V

Notes: 1. Except for the rating "Operating Temperature Range", stresses above those listed in the Table "Absolute Maximum Ratings" may cause permanent damage to the device. These are stress ratings only and operation of the device at these or any other conditions above those indicated in the Operating sections of this specification is not implied. Exposure to Absolute Maximum Rating conditions for extended periods may affect device reliability. Refer also to the SGS-THOMSON SURE Program and other relevant quality documents.
2. MIL-STD-883C, 3015.7 (100pF, 1500 Ω).
3. EIAJ IC-121 (Condition C) (200pF, 0 Ω).

DESCRIPTION (cont'd)

The ST24/25x02 are 2K bit electrically erasable programmable memories (EEPROM), organized as 256 x 8 bits. They are manufactured in SGS-THOMSON's Hi-Endurance Advanced CMOS technology which guarantees an endurance of one million erase/write cycles with a data retention of 40 years. The memories operate with a power supply value as low as 1.8V for the ST24C02R only.

Both Plastic Dual-in-Line and Plastic Small Outline packages are available.

The memories are compatible with the I^2C standard, two wire serial interface which uses a bi-directional data bus and serial clock. The memories carry a built-in 4 bit, unique device identification code (1010) corresponding to the I^2C bus definition. This is used together with 3 chip enable inputs (E2, E1, E0) so that up to 8 x 2K devices may be attached to the I^2C bus and selected individually. The memories behave as a slave device in the I^2C protocol with all memory operations synchronized by the serial clock. Read and write operations are initiated by a START condition generated by the bus master. The START condition is followed by a stream of 7 bits (identification code 1010), plus one read/write bit and terminated by an acknowledge bit.

Table 3. Device Select Code

	Device Code				Chip Enable			RW
Bit	b7	b6	b5	b4	b3	b2	b1	b0
Device Select	1	0	1	0	E2	E1	E0	RW

Note: The MSB b7 is sent first.

Table 4. Operating Modes (1)

Mode	RW bit	MODE	Bytes	Initial Sequence
Current Address Read	'1'	X	1	START, Device Select, RW = '1'
Random Address Read	'0'	X	1	START, Device Select, RW = '0', Address,
	'1'			reSTART, Device Select, RW = '1'
Sequential Read	'1'	X	1 to 256	Similar to Current or Random Mode
Byte Write	'0'	X	1	START, Device Select, RW = '0'
Multibyte Write (2)	'0'	V_{IH}	4	START, Device Select, RW = '0'
Page Write	'0'	V_{IL}	8	START, Device Select, RW = '0'

Notes: 1. X = V_{IH} or V_{IL}
2. Multibyte Write not available in ST24/25W02 versions.

When writing data to the memory it responds to the 8 bits received by asserting an acknowledge bit during the 9th bit time. When data is read by the bus master, it acknowledges the receipt of the data bytes in the same way. Data transfers are terminated with a STOP condition.

Power On Reset: V_{CC} lock out write protect. In order to prevent data corruption and inadvertent write operations during power up, a Power On Reset (POR) circuit is implemented. Until the V_{CC} voltage has reached the POR threshold value, the internal reset is active, all operations are disabled and the device will not respond to any command. In the same way, when V_{CC} drops down from the operating voltage to below the POR threshold value, all operations are disabled and the device will not respond to any command. A stable V_{CC} must be applied before applying any logic signal.

SIGNAL DESCRIPTIONS

Serial Clock (SCL). The SCL input pin is used to synchronize all data in and out of the memory. A resistor can be connected from the SCL line to V_{CC} to act as a pull up (see Figure 3).

Serial Data (SDA). The SDA pin is bi-directional and is used to transfer data in or out of the memory. It is an open drain output that may be wire-OR'ed with other open drain or open collector signals on the bus. A resistor must be connected from the SDA bus line to V_{CC} to act as pull up (see Figure 3).

Chip Enable (E2 - E0). These chip enable inputs are used to set the 3 least significant bits (b3, b2, b1) of the 7 bit device select code. These inputs may be driven dynamically or tied to V_{CC} or V_{SS} to establish the device select code.

Mode (MODE). The MODE input is available on pin 7 (see also WC feature) and may be driven dynamically. It must be at V_{IL} or V_{IH} for the Byte Write mode, V_{IH} for Multibyte Write mode or V_{IL} for Page Write mode. When unconnected, the MODE input is internally read as a V_{IH} (Multibyte Write mode).

Write Control (WC). An hardware Write Control feature (WC) is offered only for ST24W02 and ST25W02 versions on pin 7. This feature is usefull to protect the contents of the memory from any erroneous erase/write cycle. The Write Control signal is used to enable (WC = V_{IH}) or disable (WC = V_{IL}) the internal write protection. When unconnected, the WC input is internally read as V_{IL} and the memory area is not write protected.

SIGNAL DESCRIPTIONS (cont'd)

The devices with this Write Control feature no longer support the Multibyte Write mode of operation, however all other write modes are fully supported.

Refer to the AN404 Application Note for more detailed information about Write Control feature.

DEVICE OPERATION

I²C Bus Background

The ST24/25x02 support the I²C protocol. This protocol defines any device that sends data onto the bus as a transmitter and any device that reads the data as a receiver. The device that controls the data transfer is known as the master and the other as the slave. The master will always initiate a data transfer and will provide the serial clock for synchronisation. The ST24/25x02 are always slave devices in all communications.

Start Condition. START is identified by a high to low transition of the SDA line while the clock SCL is stable in the high state. A START condition must precede any command for data transfer. Except during a programming cycle, the ST24/25x02 continuously monitor the SDA and SCL signals for a START condition and will not respond unless one is given.

Stop Condition. STOP is identified by a low to high transition of the SDA line while the clock SCL is stable in the high state. A STOP condition terminates communication between the ST24/25x02 and the bus master. A STOP condition at the end of a Read command, after and only after a No Acknowledge, forces the standby state. A STOP condition at the end of a Write command triggers the internal EEPROM write cycle.

Acknowledge Bit (ACK). An acknowledge signal is used to indicate a successfull data transfer. The bus transmitter, either master or slave, will release the SDA bus after sending 8 bits of data. During the 9th clock pulse period the receiver pulls the SDA bus low to acknowledge the receipt of the 8 bits of data.

Data Input. During data input the ST24/25x02 sample the SDA bus signal on the rising edge of the clock SCL. Note that for correct device operation the SDA signal must be stable during the clock low to high transition and the data must change ONLY when the SCL line is low.

Memory Addressing. To start communication between the bus master and the slave ST24/25x02, the master must initiate a START condition. Following this, the master sends onto the SDA bus line 8 bits (MSB first) corresponding to the device select code (7 bits) and a READ or WRITE bit.

Figure 3. Maximum R_L Value versus Bus Capacitance (C_{BUS}) for an I²C Bus

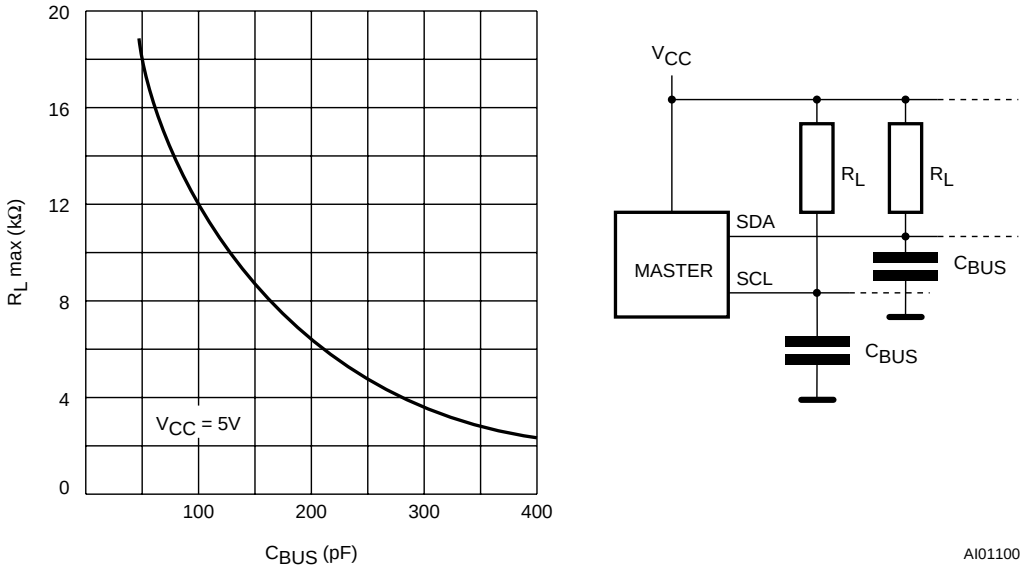


Table 5. Input Parameters (1) (T_A = 25 °C, f = 100 kHz)

Symbol	Parameter	Test Condition	Min	Max	Unit
C_{IN}	Input Capacitance (SDA)			8	pF
C_{IN}	Input Capacitance (other pins)			6	pF
Z_{WCL}	$\overline{WC}$ Input Impedance (ST24/25W02)	$V_{IN} \le 0.3\ V_{CC}$	5	20	kΩ
Z_{WCH}	$\overline{WC}$ Input Impedance (ST24/25W02)	$V_{IN} \ge 0.7\ V_{CC}$	500		kΩ
t_{LP}	Low-pass filter input time constant (SDA and SCL)			100	ns

Note: 1. Sampled only, not 100% tested.

Table 6. DC Characteristics
(T_A = 0 to 70°C, –20 to 85°C or –40 to 85°C; V_{CC} = 3V to 5.5V, 2.5V to 5.5V or 1.8V to 5.5V)

Symbol	Parameter	Test Condition	Min	Max	Unit
I_{LI}	Input Leakage Current	$0V \le V_{IN} \le V_{CC}$		±2	µA
I_{LO}	Output Leakage Current	$0V \le V_{OUT} \le V_{CC}$ SDA in Hi-Z		±2	µA
I_{CC}	Supply Current (ST24 series)	V_{CC} = 5V, f_C = 100kHz (Rise/Fall time < 10ns)		2	mA
	Supply Current (ST25 series)	V_{CC} = 2.5V, f_C = 100kHz		1	mA
I_{CC1}	Supply Current (Standby) (ST24 series)	$V_{IN} = V_{SS}$ or V_{CC}, V_{CC} = 5V		100	µA
		$V_{IN} = V_{SS}$ or V_{CC}, V_{CC} = 5V, f_C = 100kHz		300	µA
I_{CC2}	Supply Current (Standby) (ST25 series)	$V_{IN} = V_{SS}$ or V_{CC}, V_{CC} = 2.5V		5	µA
		$V_{IN} = V_{SS}$ or V_{CC}, V_{CC} = 2.5V, f_C = 100kHz		50	µA
I_{CC3}	Supply Current (Standby) (ST24C02R)	$V_{IN} = V_{SS}$ or V_{CC}, V_{CC} = 3.6V		20	µA
		$V_{IN} = V_{SS}$ or V_{CC}, V_{CC} = 3.6V, f_C = 100kHz		60	µA
I_{CC4}	Supply Current (Standby) (ST24C02R)	$V_{IN} = V_{SS}$ or V_{CC}, V_{CC} = 1.8V		10	µA
		$V_{IN} = V_{SS}$ or V_{CC}, V_{CC} = 1.8V, f_C = 100kHz		20	µA
V_{IL}	Input Low Voltage (SCL, SDA)		–0.3	0.3 V_{CC}	V
V_{IH}	Input High Voltage (SCL, SDA)		0.7 V_{CC}	V_{CC} + 1	V
V_{IL}	Input Low Voltage (E0-E2, MODE, $\overline{WC}$)		–0.3	0.5	V
V_{IH}	Input High Voltage (E0-E2, MODE, $\overline{WC}$)		V_{CC} – 0.5	V_{CC} + 1	V
V_{OL}	Output Low Voltage (ST24 series)	I_{OL} = 3mA, V_{CC} = 5V		0.4	V
	Output Low Voltage (ST25 series)	I_{OL} = 2.1mA, V_{CC} = 2.5V		0.4	V
	Output Low Voltage (ST24C02R)	I_{OL} = 1mA, V_{CC} = 1.8V		0.3	V

Table 7. AC Characteristics
(T_A = 0 to 70°C, –20 to 85°C or –40 to 85°C; V_{CC} = 3V to 5.5V, 2.5V to 5.5V or 1.8V to 5.5V)

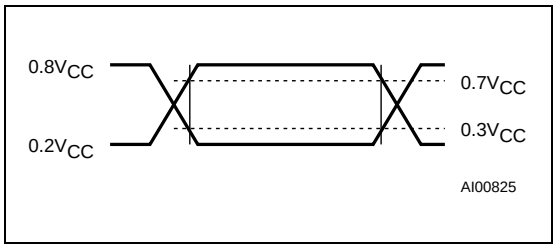
Symbol	Alt	Parameter	Min	Max	Unit
t_{CH1CH2}	t_R	Clock Rise Time		1	µs
t_{CL1CL2}	t_F	Clock Fall Time		300	ns
t_{DH1DH2}	t_R	Input Rise Time		1	µs
t_{DL1DL1}	t_F	Input Fall Time		300	ns
t_{CHDX} (1)	$t_{SU:STA}$	Clock High to Input Transition	4.7		µs
t_{CHCL}	t_{HIGH}	Clock Pulse Width High	4		µs
t_{DLCL}	$t_{HD:STA}$	Input Low to Clock Low (START)	4		µs
t_{CLDX}	$t_{HD:DAT}$	Clock Low to Input Transition	0		µs
t_{CLCH}	t_{LOW}	Clock Pulse Width Low	4.7		µs
t_{DXCX}	$t_{SU:DAT}$	Input Transition to Clock Transition	250		ns
t_{CHDH}	$t_{SU:STO}$	Clock High to Input High (STOP)	4.7		µs
t_{DHDL}	t_{BUF}	Input High to Input Low (Bus Free)	4.7		µs
t_{CLQV} (2)	t_{AA}	Clock Low to Next Data Out Valid	0.3	3.5	µs
t_{CLQX}	t_{DH}	Data Out Hold Time	300		ns
f_C	f_{SCL}	Clock Frequency		100	kHz
t_W (3)	t_{WR}	Write Time		10	ms

Notes: 1. For a reSTART condition, or following a write cycle.
2. The minimum value delays the falling/rising edge of SDA away from SCL = 1 in order to avoid unwanted START and/or STOP conditions.
3. In the Multibyte Write mode only, if accessed bytes are on two consecutive 8 bytes rows (6 address MSB are not constant) the maximum programming time is doubled to 20ms.

AC MEASUREMENT CONDITIONS

Input Rise and Fall Times	≤ 50ns
Input Pulse Voltages	$0.2V_{CC}$ to $0.8V_{CC}$
Input and Output Timing Ref. Voltages	$0.3V_{CC}$ to $0.7V_{CC}$

Figure 4. AC Testing Input Output Waveforms



DEVICE OPERATION (cont'd)

The 4 most significant bits of the device select code are the device type identifier, corresponding to the I^2C bus definition. For these memories the 4 bits are fixed as 1010b. The following 3 bits identify the specific memory on the bus. They are matched to the chip enable signals E2, E1, E0. Thus up to 8 x 2K memories can be connected on the same bus giving a memory capacity total of 16K bits. After a START condition any memory on the bus will identify the device code and compare the following 3 bits to its chip enable inputs E2, E1, E0.

The 8th bit sent is the read or write bit ($R\overline{W}$), this bit is set to '1' for read and '0' for write operations. If a match is found, the corresponding memory will acknowledge the identification on the SDA bus during the 9th bit time.

Figure 5. AC Waveforms

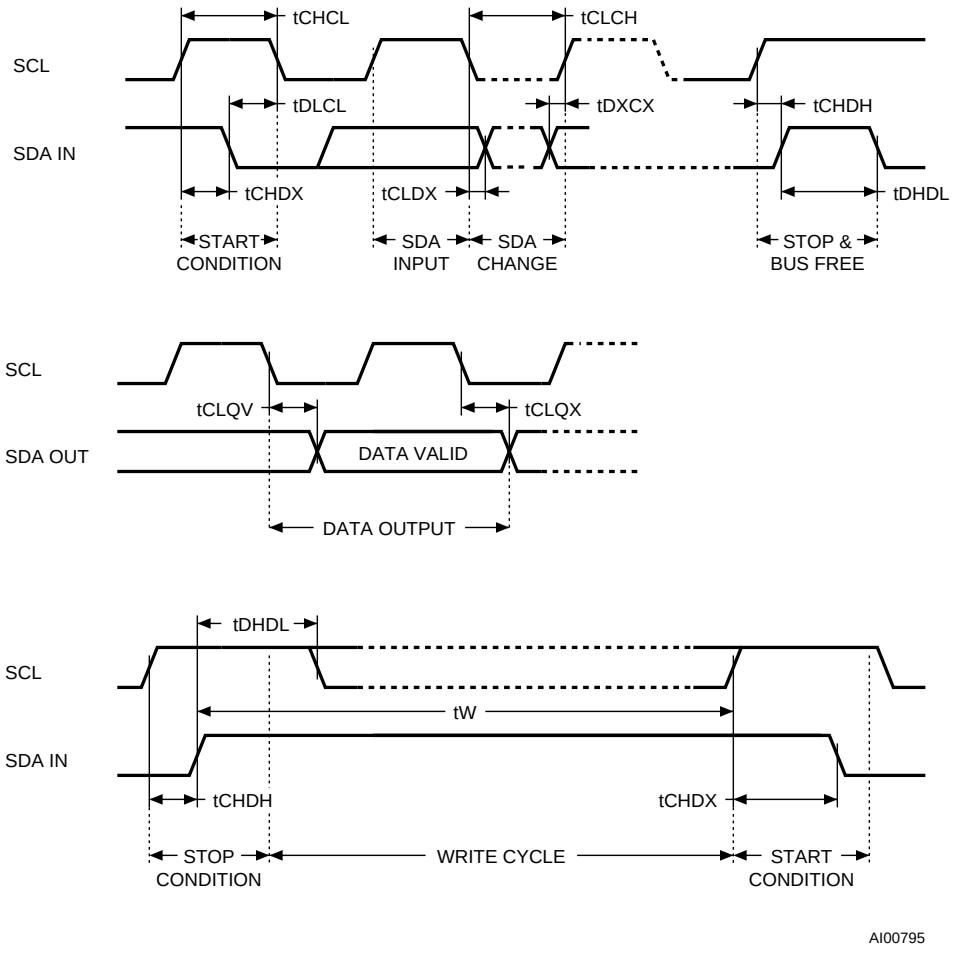
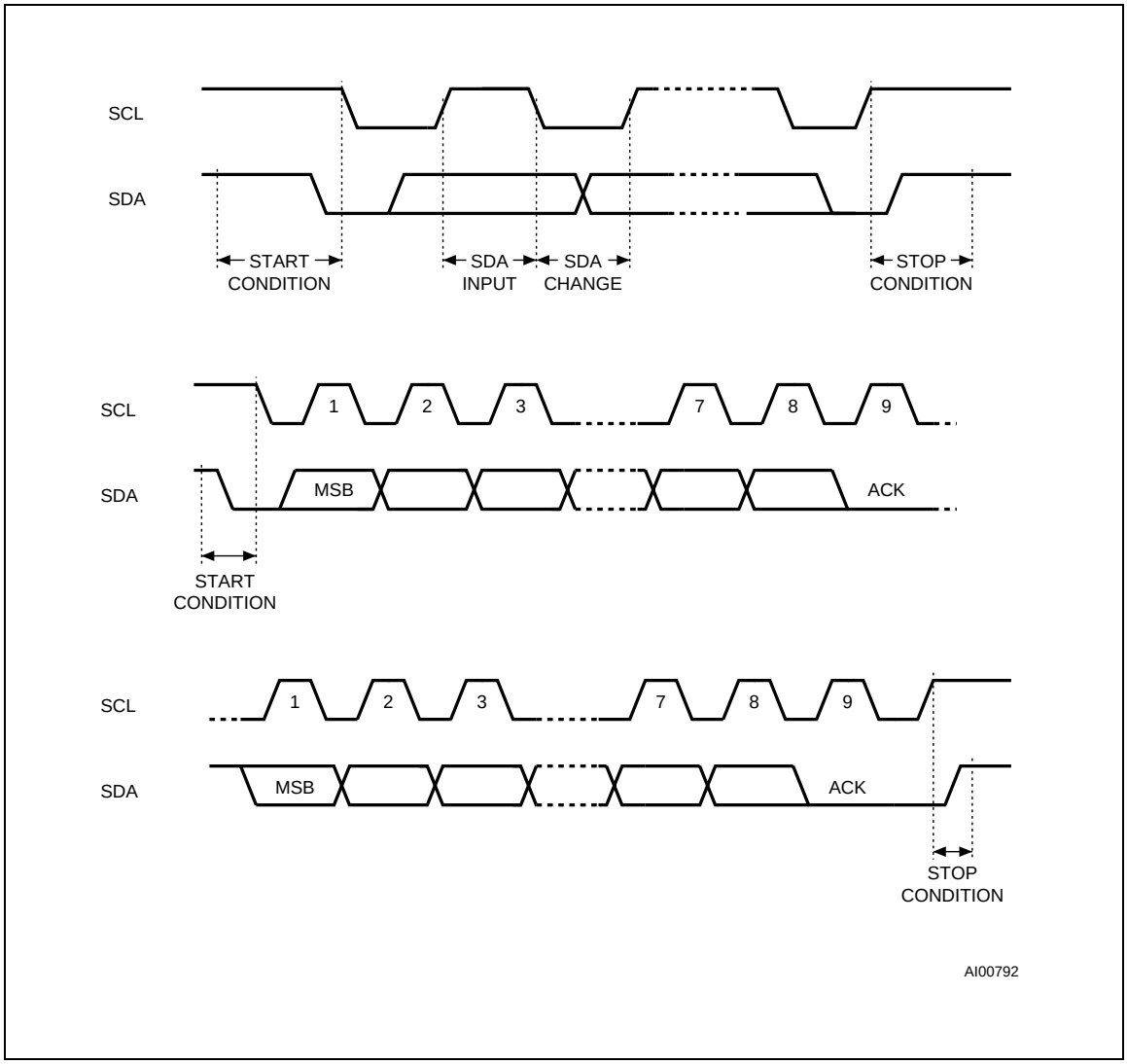


Figure 6. I²C Bus Protocol



Write Operations

The Multibyte Write mode (only available on the ST24/25C02 and ST24C02R versions) is selected when the MODE pin is at V_{IH} and the Page Write mode when MODE pin is at V_{IL}. The MODE pin may be driven dynamically with CMOS input levels.

Following a START condition the master sends a device select code with the $\overline{RW}$ bit reset to '0'. The memory acknowledges this and waits for a byte address. The byte address of 8 bits provides access to 256 bytes of the memory. After receipt of the byte address the device again responds with an acknowledge.

For the ST24/25W02 versions, any write command with $\overline{WC}$ = 1 will not modify the memory content.

Byte Write. In the Byte Write mode the master sends one data byte, which is acknowledged by the memory. The master then terminates the transfer by generating a STOP condition. The Write mode is independant of the state of the MODE pin which could be left floating if only this mode was to be used. However it is not a recommended operating mode, as this pin has to be connected to either V_{IH} or V_{IL}, to minimize the stand-by current.

Multibyte Write. For the Multibyte Write mode, the MODE pin must be at V_{IH}. The Multibyte Write mode can be started from any address in the memory. The master sends from one up to 4 bytes of data, which are each acknowledged by the memory. The transfer is terminated by the master generating a STOP condition. The duration of the write cycle is t_W = 10ms maximum except when bytes are accessed on 2 rows (that is have different values for the 6 most significant address bits A7-A2), the programming time is then doubled to a maximum of 20ms. Writing more than 4 bytes in the Multibyte Write mode may modify data bytes in an adjacent row (one row is 8 bytes long). However, the Multibyte Write can properly write up to 8 consecutive bytes only if the first address of these 8 bytes is the first address of the row, the 7 following bytes being written in the 7 following bytes of this same row.

Page Write. For the Page Write mode, the MODE pin must be at V_{IL}. The Page Write mode allows up to 8 bytes to be written in a single write cycle, provided that they are all located in the same 'row' in the memory: that is the 5 most significant memory address bits (A7-A3) are the same. The master sends from one up to 8 bytes of data, which are each acknowledged by the memory. After each byte is transfered, the internal byte address counter (3 least significant bits only) is incremented. The transfer is terminated by the master generating a STOP condition. Care must be taken to avoid address counter 'roll-over' which could result in data being overwritten. Note that, for any write mode, the generation by the master of the STOP condition starts the internal memory program cycle. All inputs are disabled until the completion of this cycle and the memory will not respond to any request.

Figure 7. Write Cycle Polling using ACK

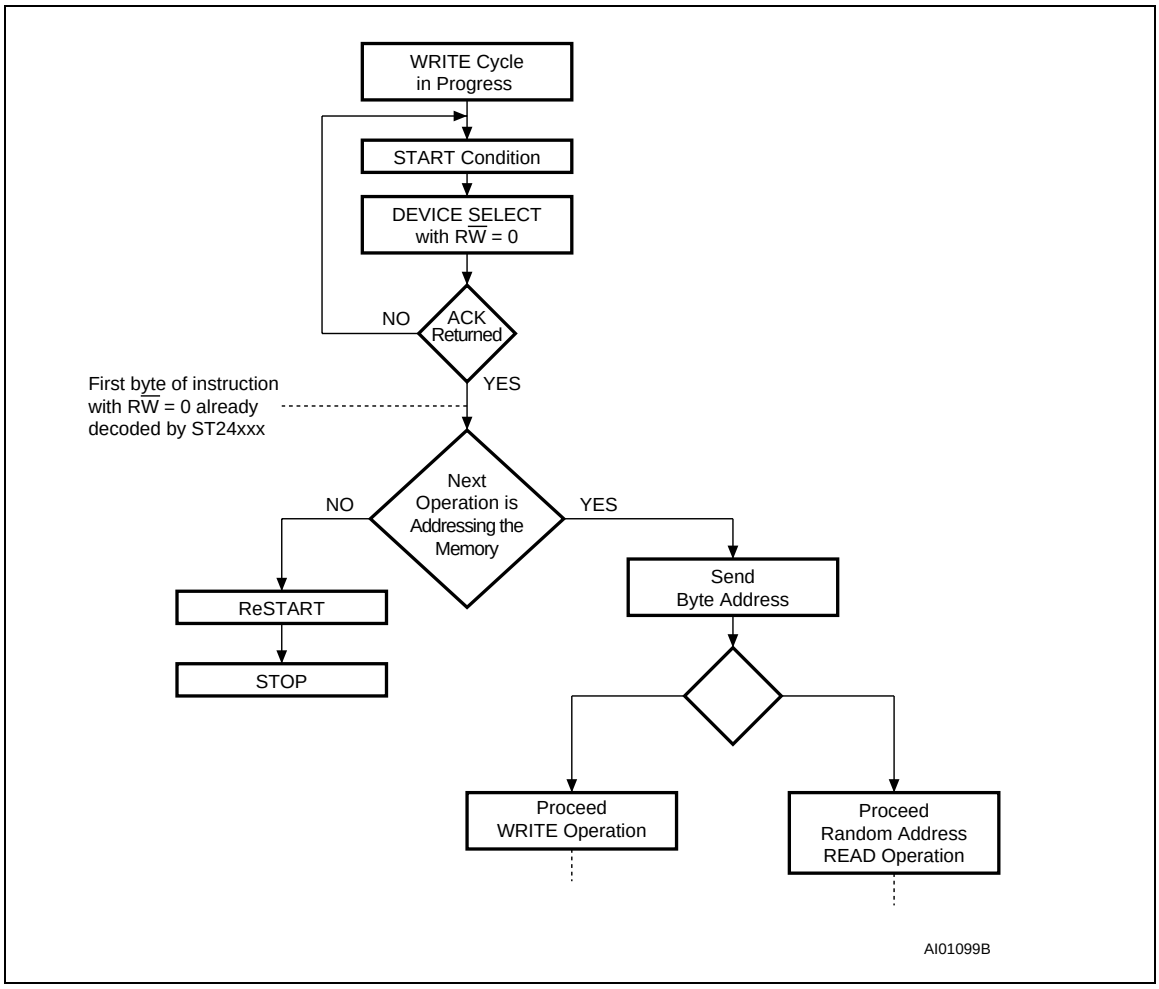
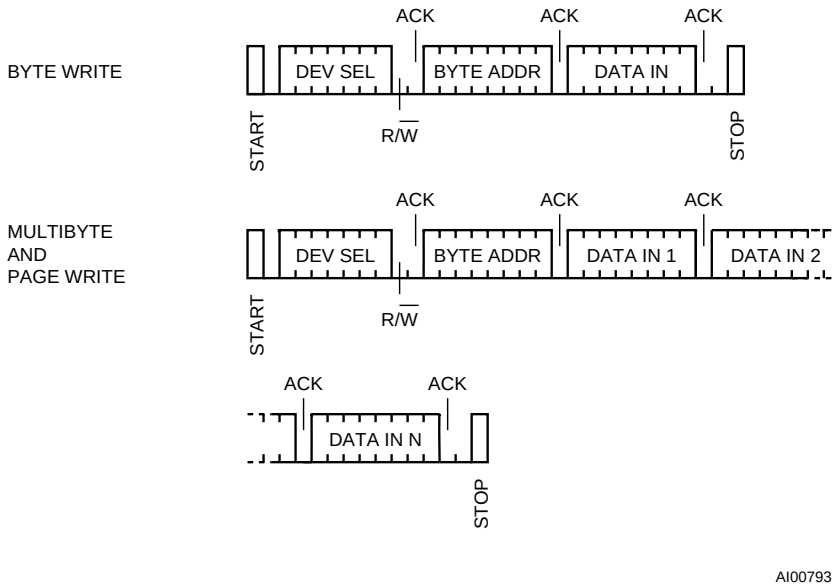


Figure 8. Write Modes Sequence (ST24/25C02 and ST24C02R)



Minimizing System Delays by Polling On ACK. During the internal write cycle, the memory disconnects itself from the bus in order to copy the data from the internal latches to the memory cells. The maximum value of the write time (t_W) is given in the AC Characteristics table, since the typical time is shorter, the time seen by the system may be reduced by an ACK polling sequence issued by the master. The sequence is as follows:

– Initial condition: a Write is in progress (see Figure 7).
– Step 1: the master issues a START condition followed by a device select byte (1st byte of the new instruction).
– Step 2: if the memory is busy with the internal write cycle, no ACK will be returned and the master goes back to Step 1. If the memory has terminated the internal write cycle, it will respond with an ACK, indicating that the memory is ready to receive the second part of the next instruction (the first byte of this instruction was already sent during Step 1).

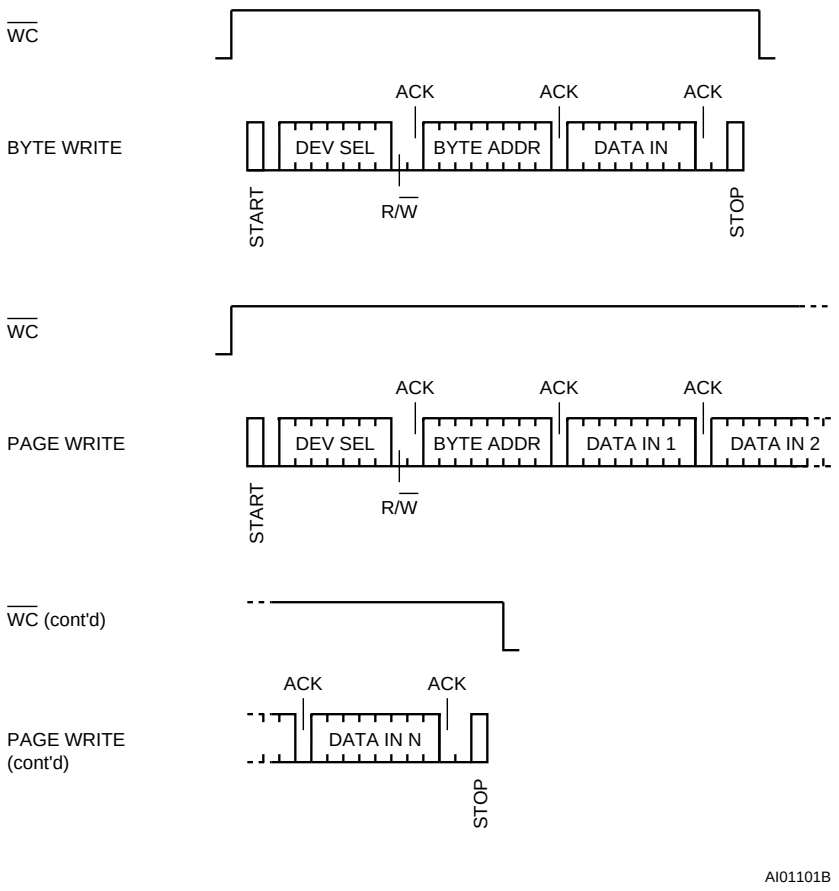
Read Operations

Read operations are independent of the state of the MODE pin. On delivery, the memory content is set at all "1's" (or FFh).

Current Address Read. The memory has an internal byte address counter. Each time a byte is read, this counter is incremented. For the Current Address Read mode, following a START condition, the master sends a memory address with the $\overline{RW}$ bit set to '1'. The memory acknowledges this and outputs the byte addressed by the internal byte address counter. This counter is then incremented. The master does NOT acknowledge the byte output, but terminates the transfer with a STOP condition.

Random Address Read. A dummy write is performed to load the address into the address counter, see Figure 10. This is followed by another START condition from the master and the byte address is repeated with the $\overline{RW}$ bit set to '1'. The memory acknowledges this and outputs the byte addressed. The master have to NOT acknowledge the byte output, but terminates the transfer with a STOP condition.

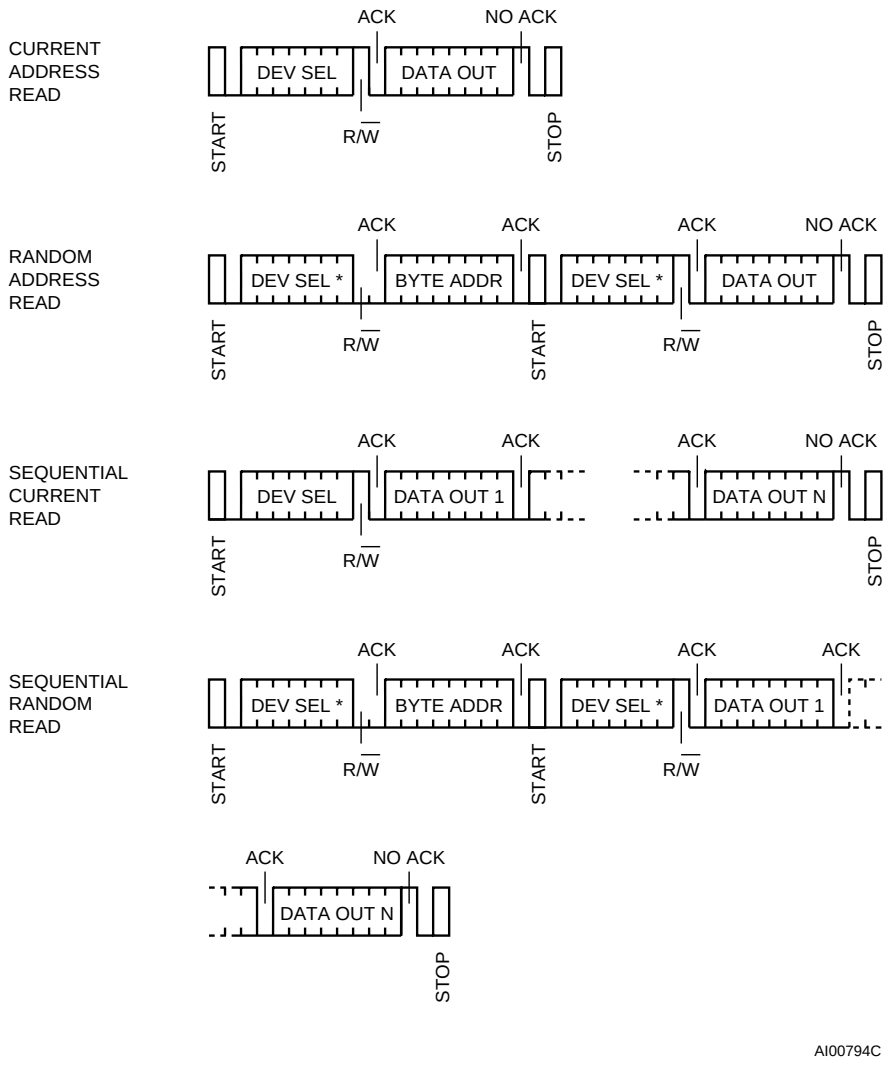
Figure 9. Write Modes Sequence with Write Control = 1 (ST24/25W02)



Sequential Read. This mode can be initiated with either a Current Address Read or a Random Address Read. However, in this case the master DOES acknowledge the data byte output and the memory continues to output the next byte in sequence. To terminate the stream of bytes, the master must NOT acknowledge the last byte output, but MUST generate a STOP condition. The output data is from consecutive byte addresses, with the internal byte address counter automatically incremented after each byte output. After a count of the last memory address, the address counter will 'roll- over' and the memory will continue to output data.

Acknowledge in Read Mode. In all read modes the ST24/25x02 wait for an acknowledge during the 9th bit time. If the master does not pull the SDA line low during this time, the ST24/25x02 terminate the data transfer and switches to a standby state.

Figure 10. Read Modes Sequence



Note: * The 7 Most Significant bits of DEV SEL bytes of a Random Read (1st byte and 3rd byte) must be identical.

ORDERING INFORMATION SCHEME

Example: ST24C02 M 1 TR

Operating Voltage		Range
ST24C02	3V to 5.5V	Standard
ST24W02	3V to 5.5V	Hardware Write Control
ST25C02	2.5V to 5.5V	Standard
ST25W02	2.5V to 5.5V	Hardware Write Control
ST24C02R	1.8V to 5.5V	Standard

Package	
B	PSDIP8 0.25mm Frame
M	SO8 150mil Width

Temperature Range	
1	0 to 70 °C
5 *	–20 to 85 °C
6	–40 to 85 °C
3 *	–40 to 125 °C

Option	
TR	Tape & Reel Packing

Notes: 3 * Temperature range on special request only.
5 * Temperature range for ST24C02R only.

Parts are shipped with the memory content set at all "1's" (FFh).

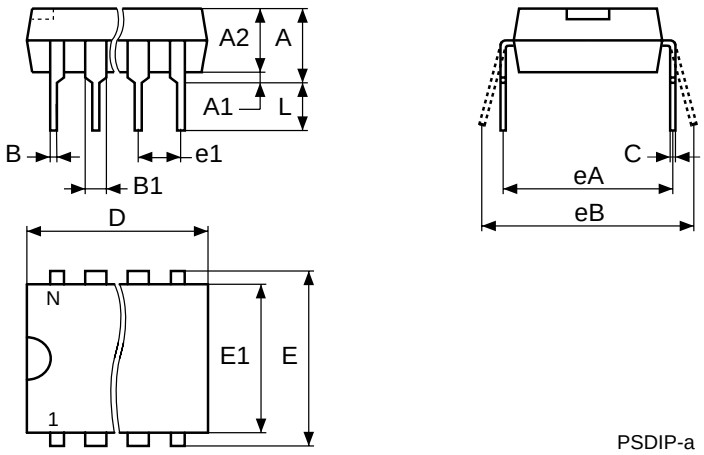
For a list of available options (Operating Voltage, Range, Package, etc...) refer to the current Memory Shortform catalogue.

For further information on any aspect of this device, please contact the SGS-THOMSON Sales Office nearest to you.

PSDIP8 - 8 pin Plastic Skinny DIP, 0.25mm lead frame

Symb	mm			inches		
	Typ	Min	Max	Typ	Min	Max
A		3.90	5.90		0.154	0.232
A1		0.49	–		0.019	–
A2		3.30	5.30		0.130	0.209
B		0.36	0.56		0.014	0.022
B1		1.15	1.65		0.045	0.065
C		0.20	0.36		0.008	0.014
D		9.20	9.90		0.362	0.390
E	7.62	–	–	0.300	–	–
E1		6.00	6.70		0.236	0.264
e1	2.54	–	–	0.100	–	–
eA		7.80	–		0.307	–
eB			10.00			0.394
L		3.00	3.80		0.118	0.150
N	8			8		

PSDIP8



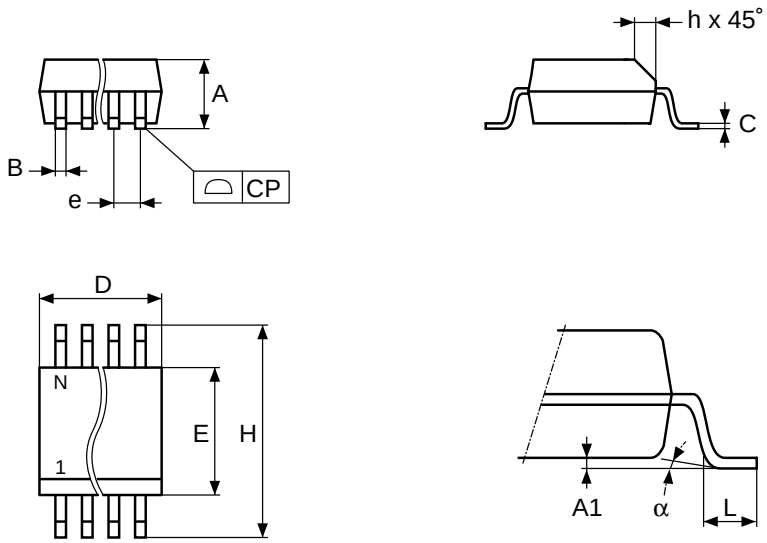
PSDIP-a

Drawing is not to scale

SO8 - 8 lead Plastic Small Outline, 150 mils body width

Symb	mm			inches		
	Typ	Min	Max	Typ	Min	Max
A		1.35	1.75		0.053	0.069
A1		0.10	0.25		0.004	0.010
B		0.33	0.51		0.013	0.020
C		0.19	0.25		0.007	0.010
D		4.80	5.00		0.189	0.197
E		3.80	4.00		0.150	0.157
e	1.27	–	–	0.050	–	–
H		5.80	6.20		0.228	0.244
h		0.25	0.50		0.010	0.020
L		0.40	0.90		0.016	0.035
α		0°	8°		0°	8°
N	8			8		
CP			0.10			0.004

SO8



SO-a

Drawing is not to scale

HOCHSCHULE
RHEIN-WAAL
Rhine-Waal University
of Applied Sciences

LABORATORY

Microelectronic Control Systems

3

EXPERIMENT:

Levitating Magnet

1 The experiment set-up

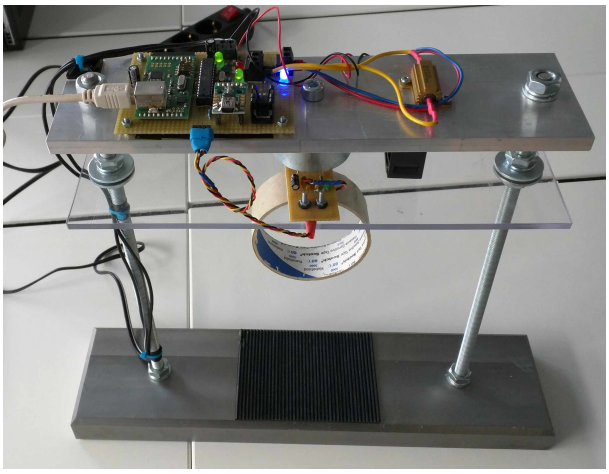


Figure 1: Experiment Setup

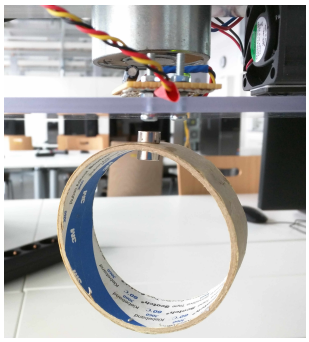


Figure 2: Levitating Magnet

The experiment setup is made with heavy metal. Be careful when moving it. Do not let it fall down.

In this experiment there is one permanent magnet that should levitate in a stable position in the air, see Figure 1 and 2. To be able to do this the gravity force must have an oppose force. In this experimental setup, a fixed permanent magnet is used to rise the levitating magnet. Only these two forces would be very unstable. So there is a hall effect sensor and an electric-magnet. The hall effect sensor is the position sensor and the electric-magnet will push the levitating magnet down. The electromagnet has approximate the same power like the fixed magnet.

So we have:

- Actor: Electromagnet
- Sensor: Hall effect sensor

The hall effect sensor will measure all magnetic fields. Therefore, the magnetic fields generated by the fixed magnet and the electromagnet will also be measured. It it your part to write a software controller that calculates the true value for the levitating magnet only.

A schematic drawing of the mechanical part is in Figure 3.

1.1 Electronic

The electronic part is shown in Figure 4. Please have a look at the schematic diagram. The electric magnet is controlled with an PWM signal.

1.2 The hall effect sensor

You can find the datasheet of this sensor here: http://www.mouser.com/ds/2/187/honeywell-sensing-ss490%20series-solidstate-product--947420.pdf

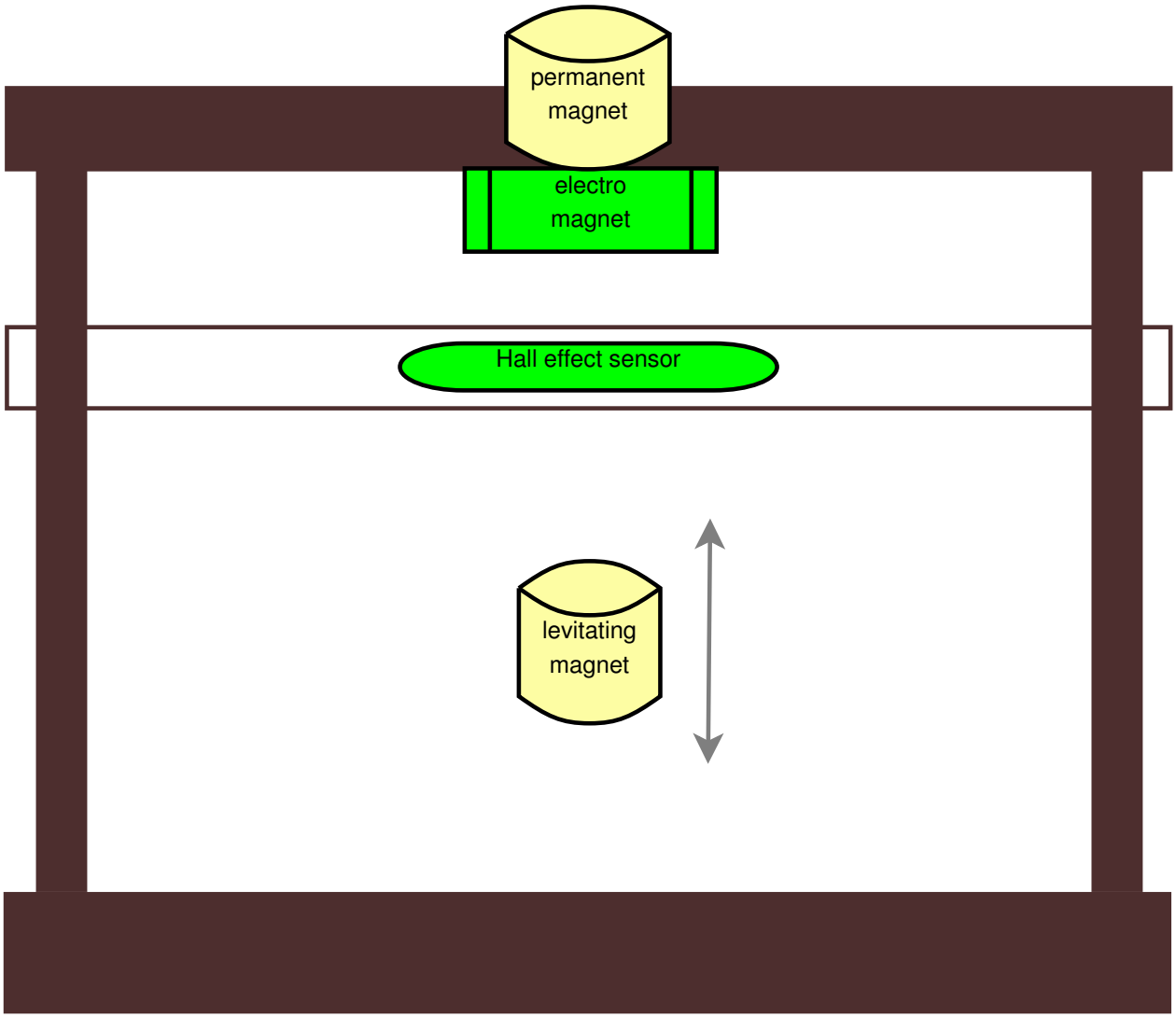


Figure 3: The mechanical part

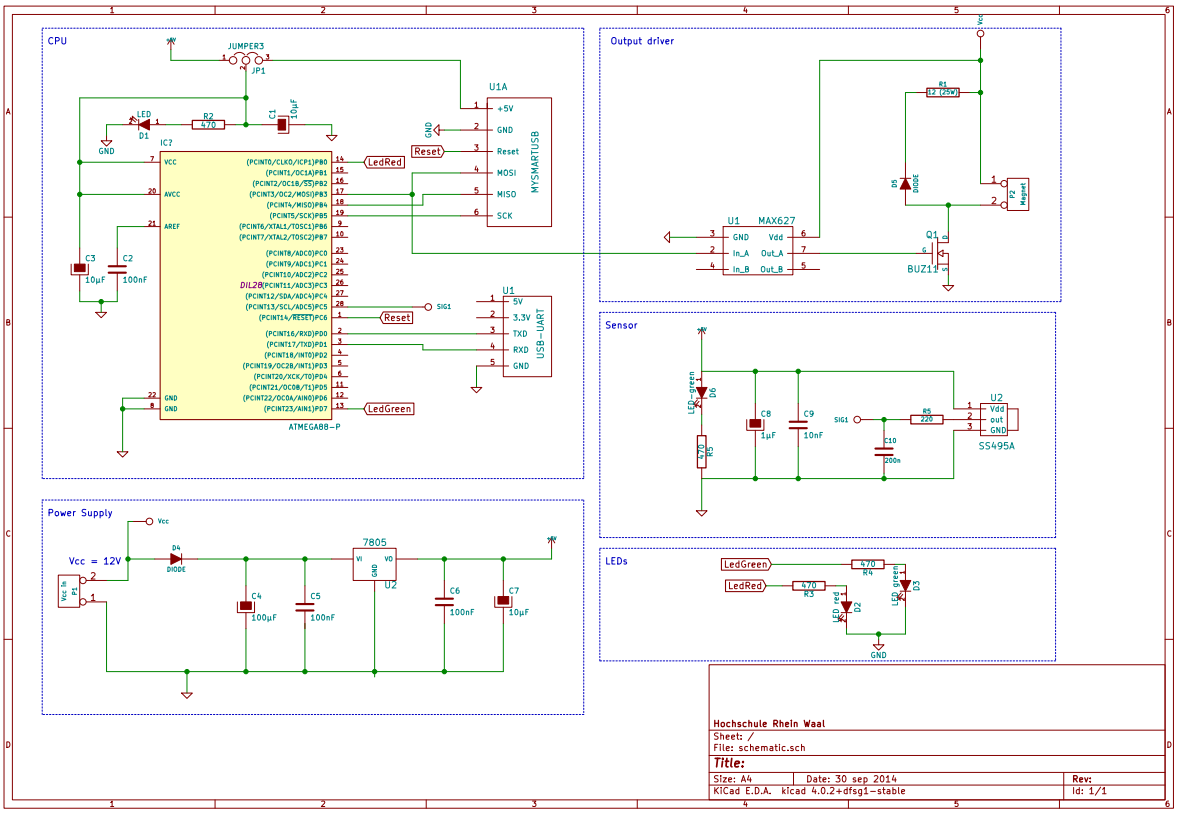


Figure 4: The electronic

2 The challenge

Task 1:
The challenge is to write a program, that holds the magnet in a levitating position. The magnet should not fall down or move to the top.

You will get a few problems:

- The hall effect sensor is affected by the electromagnet. To solve this, the first step is to write a function to calculate the position of the magnet. An automatic calibrate function could be useful.
- The controller must react very fast, so write efficient code
- You need to find good controller parameter.

Task 2 (Optional):
If your controller program works fine, try to use the serial communication to set the position of the levitating magnet. It it possible change the position of the levitating magnet up and down for 2mm.

3 Appendix

Here are some additional information's.

3.1 Microcontroller

The microcontroller is the same type like the one from the MyAVR board. Different is here the clock rate, because this experiment needs a very fast reaction from the microcontroller.

- Type: Atmel Mega88
- Clock rate: 20MHz

3.2 Magnetic force

The magnetic force is:

$$F = \frac{1}{4\pi\mu_0} \frac{p_1 * p_2}{r^2}$$

The left part $\frac{1}{4\pi\mu_0}$ is a constant value. The right part has p_1 and p_2. In this experiment the controller can reduce p_1 to approx zero, when the electromagnet is constant on. The field of the electric magnet neutralizes the field of the fixed permanent magnet!

p_2 is the levitating magnet and a constant value.

The distance r has a squared effect!

3.3 Hints

- Use the red and green LED for debugging
- You can use the prescaler 64 for the ADC (overclocking), when only the highest 8 bit are used
- Use the free running mode for the ADC and execute your controller for every complete ADC result
- Use the maximum PWM speed that is possible (prescaler = 1), use Fast-PWM with 8 bit

HOCHSCHULE
RHEIN-WAAL
Rhine-Waal University
of Applied Sciences

LABORATORY

Microelectronic Control Systems

1

EXPERIMENT:

Retake Microcontroller Programming

1 The Lab

In the lab are 20 workplaces with a personal computer and a MyAVR board. The MyAVR board is connected and powered via USB (Universal Serial Bus). The microcontroller used in this lab is a Atmel Mega88pa. To get more information and to download the datasheet please visit http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf

1.1 MyAVR board

The MyAVR board has the basic input and output devices:

- LEDs
- Keys
- Potentiometer
- Sounder
- LCD display

The ports B, C, and D are used to connect to I/O (Input/Output) devices using wires (example shown in Figure 1).

1.2 Software

The used software is complete free and open source. You can download these software using the links provided below:

- Operating System: Ubuntu http://www.ubuntu.com
- Editor: Gedit, VI, VIM, Geany (use what you like)
- Compiler: avr-gcc http://gcc.gnu.org
- Programmer: avrdude http://www.nongnu.org/avrdude

To build a program and flash it to a microcontroller it is possible to execute every step manually. However, it is very time consuming. A more convenient way to do this process is using makefiles.

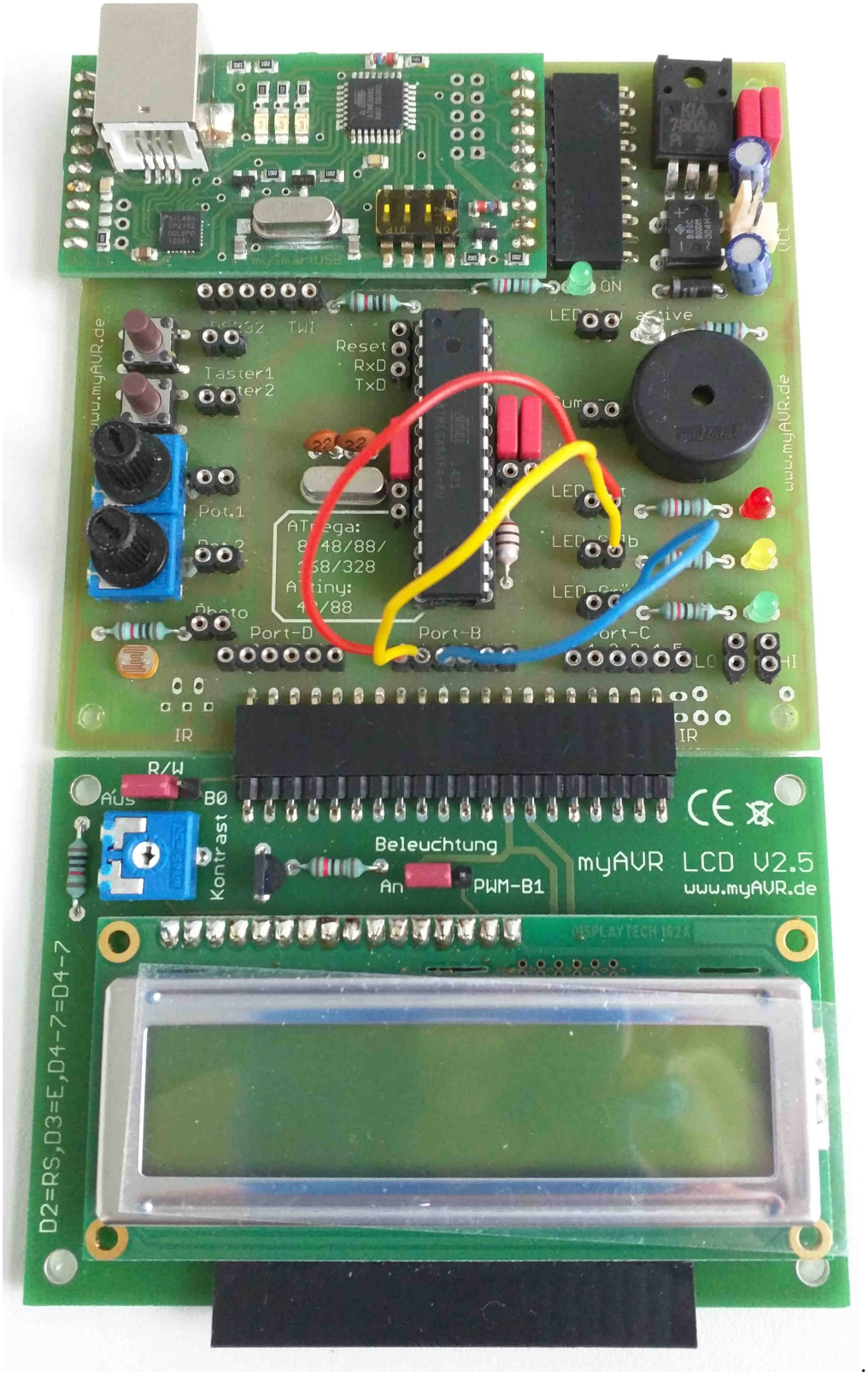


Figure 1: My AVR board with wires

2 Tasks

To retake the microcontroller lab here are some tasks that contain all the important items. It only makes sense to work on these tasks if you have done the microcontroller labs.

The first step is to compile and flash a pre written program. All files you need are in the directory Templates/Retake.

Open a terminal (STRG + ALT + T) and enter *cd Templates/Retake*.

Task 1:
To complete the first step, enter *make flash1*. The LCD display should now show the text: "first step ok".

2.1 Port I/O

Lets use the digital I/O. Connect:

- Key 1 to PB1
- Key 2 to PB0
- The red LED to PC0

Task 2:
With this configuration, flash program two (**make flash2**). With the two buttons you can now enable or disable the LED.

Read and understand the simple source files:

- two.c
- init.c

The following files are for your code:

- three.c: Task in 2.1
- four.c: Task in 2.2
- five.c: Task in 2.3
- six.c: Task in 2.4
- seven.c: Task in 2.5
- eight.c: Task in 2.6

Task 3:
Add more wires and edit the file three.c to perform these functions:

- Key 1 enables the red LED
- Key 2 enables the green LED
- When both Keys are pressed, the red and green LED are disabled
- The yellow LED is on, when any key is pressed, otherwise off

You will get a problem. It is not possible to release both buttons at the same time. So add a line to wait until both buttons are released.

2.2 Simple delay

Sometimes it is necessary to wait a short time. A simple solution is the delay function. Include the program library <util/delay.h> and use the function **_delay_ms(uint8_t time);**.

Task 4:
Write a program that can enable and disable all LEDs (at the same time) with only one key. You will need a short delay for debouncing.

2.3 Analog input

Task 5:
Connect a potentiometer to one of the analog inputs from the microcontroller. Read the value from the analog input and show it on the LCD display.

To get help how the analog to digital converter works, read the datasheet.

Hint: The LCD display can only show strings. To show a number on the screen, you need to generate a string that contains this number. You can use sprintf.

2.4 Timer / Counter

The Atmel Mega88 has three Timer/Counter. They can be used for:

- Time measurement
- Cyclical executions (interrupt)
- PWM generation (can be done by the hardware)
- Counting

Task 6:
Write a program to flash one LED with 1 Hz with a Timer/Counter. Find out the best Timer/Counter and prescaler settings to do this. The CPU speed is 8 MHz.

2.5 Interrupts

Task 7:
Interrupts are important and often used. Expand the flashing LED program, that a second LED flashes faster and the third LED is controlled with the buttons. Use only interrupts to realize this.

Please disconnect the LCD display and remove all LCD functions to use the external interrupts.

2.6 I^2C Bus

The I^2C Bus is a often used data bus for chip to chip communication. The Atmel Mega88 has an internal hardware driver to use this bus. Read the paged 230 to 233 of the datasheet.

Task 8:
Write a program that checks if a device is connected to the bus.
Connect:

- PC0: Red LED
- PC1: Green LED

Device address:

- EEPROM (All jumpers are set to zero): 0xA0
- DS1307: 0xD0

To test your program, connect and disconnect the TWI add-on modules. The LED with the corresponding color should only be on, if the TWI device is connected:

- Red LED: EEPROM
- Green LED: DS1307

2.7 I²C Real time clock

Task 9 (Optional):
Use the RTC module and the LCD to create a simple clock. The LCD display should show the current time and date. Create the file nine.c for that on your own and edit the makefile to build and upload the program.

HOCHSCHULE
RHEIN-WAAL
Rhine-Waal University
of Applied Sciences

LABORATORY

Microelectronic Control Systems

2

EXPERIMENT:

Rotor

1 The experiment set-up

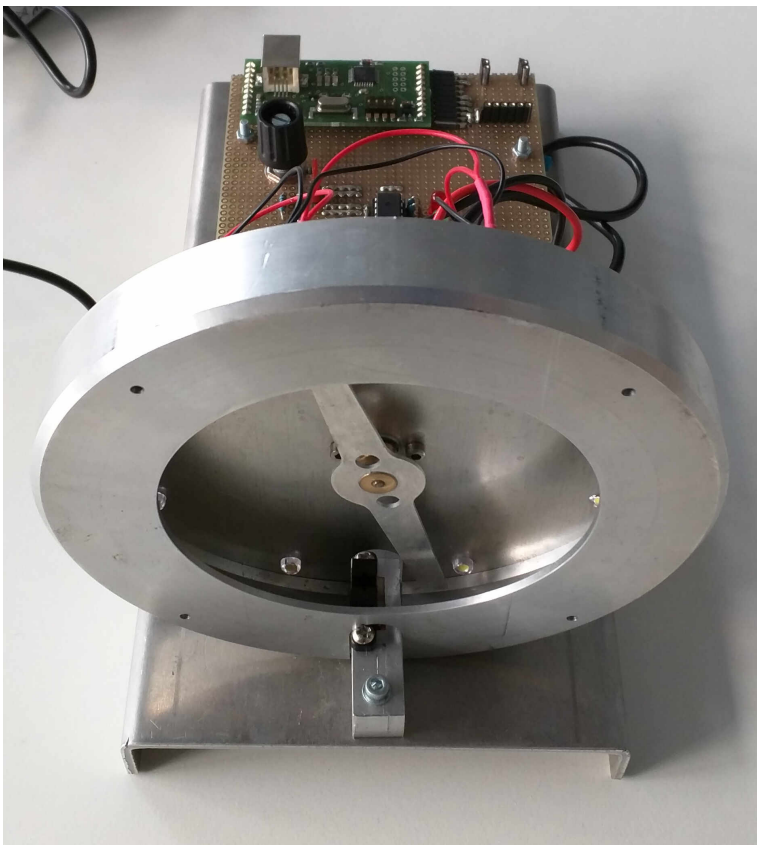


Figure 1: The experiment setup

In this experiment there is a rotor, that is directly mounted on the axis of a DC motor. This rotor is inside a case with LED illumination, see Figure 1. The DC motor can be regulated with a PWM signal. The LEDs can be switched on and off.

So we have:

- Actors:
  - LEDs
  - DC motor with PWM speed control
- Sensor: one light barrier that will be interrupted two times per rotation

A schematic drawing of the mechanical part is in Figure 2.

When the LEDs are flashing with the same frequency of the rotation, it looks like that the rotor stands still, this is called the stroboscopic effect. The human eyes will only see the moments when the light is on.

You can find more information about this effec here: https://en.wikipedia.org/wiki/Stroboscopic_effect

Hochschule Rhein Waal
Marie-Curie-Straße 1
D-47533 Kleve

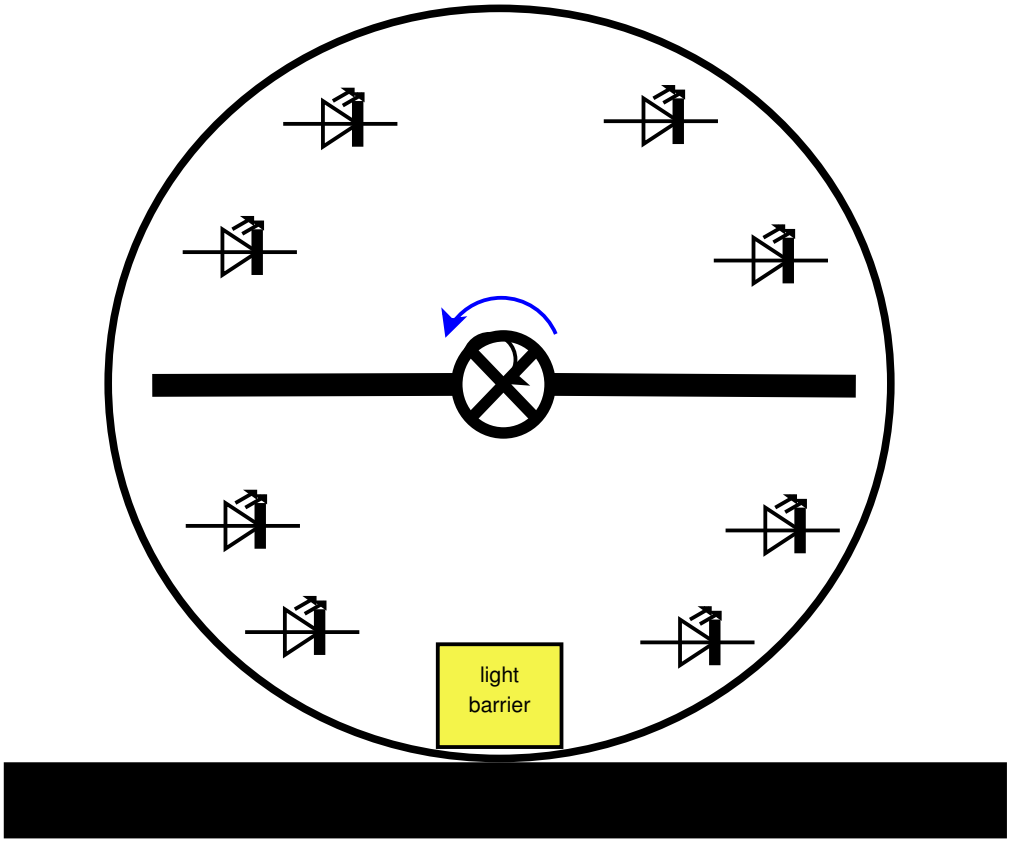


Figure 2: The mechanical part

1.1 Electronic

The electronic circuit is shown in Figure 3. Please take a look at this schematic diagram. The two LEDs (red and green) at the ports PB0 and PD7 are for software debugging. You can use this LEDs like you want. The USB to UART part is for add-ons you will not need it for this experiment.

Prepare 1: Write down the used ports for the input and output.

Prepare 2: Between the potentiometer and the microcontroller is the resistor R1. This resistor has no effect to the analog value. What could be the reason that there is a resistor?

Prepare 3: The filter FB1 protects the microcontroller against high frequency oscillation from the motor. What effect does this filter have on the response characteristic of the motor?

Hochschule Rhein Waal
Marie-Curie-Straße 1
D-47533 Kleve

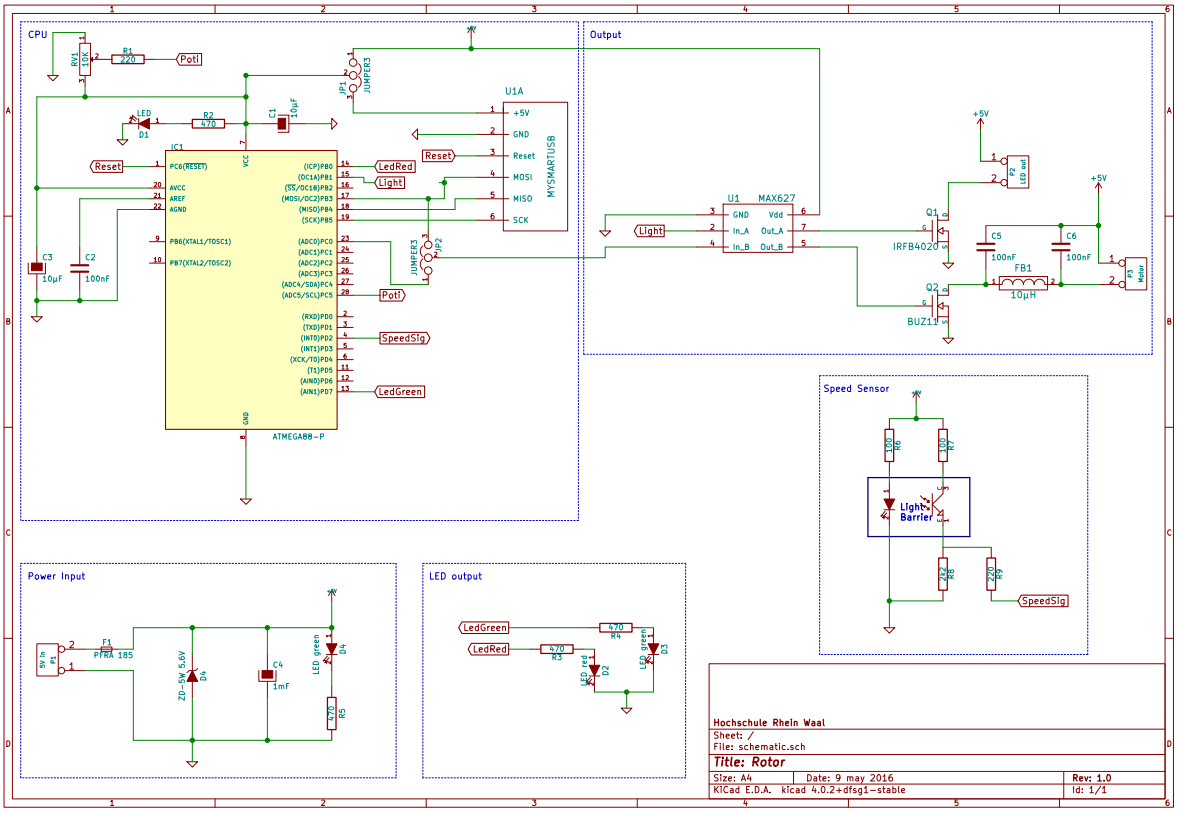


Figure 3: The electronic circuit

Hochschule Rhein Waal
Marie-Curie-Straße 1
D-47533 Kleve

2 The challenge

There are two different challenges here!

Task 1:
The fist challenge is to write a program, that flashes the LEDs to make it look like the rotor is standing. The rotation speed can be adjusted with the potentiometer.

Here are the steps to solve:

1. Write a program to get the reading of the potentiometer
2. Use this value to create a PWM signal to set the rotation speed
3. Write a function to measure the rotation speed with the light barrier
4. Let the LEDs flash a short time when the rotor is vertical
5. Then let the LEDs flash a short time when the rotor is horizontal

Task 2:
The second challenge is to write a program, that regulates the rotation speed with an PWM signal that it always looks like the rotor is standing. The flashing speed should be adjustable with a potentiometer.
Read the value from the potentiometer to adjust the flashing frequency from 50Hz to 200Hz. The controller should react when anybody changes the flashing speed and adapt the rotation speed.

Here are the steps to solve:

1. Write a program to get the reading of the potentiometer
2. Use this value to create a flashing signal for the LEDs. The on-time should be 5 percent.
3. Write a function to create the PWM signal for the motor
4. Write a function to measure the rotation speed
5. Create the software controller

2.1 Hints

- To let the LEDs flash when the rotor is horizontal the flash must be half a period later than the flash at the vertical position
- To create PWM by software you can use the overflow interrupt of a free Timer/Counter
- The PWM frequency for the motor should be at 1kHz

Hochschule Rhein Waal
Marie-Curie-Straße 1
D-47533 Kleve

2.2 Motor check

Task 3 (Optional):
Add a detection for a blocked motor. If the motor is not able to turn for more than three seconds, your program should disable the motor and flash the LED's with $1Hz$.

3 Appendix

Here are some additional information.

3.1 Microcontroller

The used microcontroller in this experiment is from Atmel:

- Type: Atmel Mega88PA
- Clock rate: 8MHz

To get more information read: http://www.atmel.com/devices/atmega88.aspx

Hochschule Rhein Waal
Marie-Curie-Straße 1
D-47533 Kleve