

# Exercises pool

## LVA Microcontroller 182.694

Revision 0.996 S

### Recommended exercises

	Part I – ASM	Part II – C	Part III – TinyOS
<b>Week 1</b>	<a href="#">1.1</a> , <a href="#">2.1.1</a> , <a href="#">2.1.2</a> , <a href="#">2.2.1</a>	TBA	TBA
<b>Week 2</b>	<a href="#">2.2.2</a> , <a href="#">2.2.4</a> , <a href="#">2.2.5</a> , <a href="#">2.4.1</a>		
<b>Week 3</b>	<a href="#">2.2.8</a> , <a href="#">2.4.2</a> , <a href="#">2.4.3</a> , <a href="#">2.4.4</a>		
<b>Week 4</b>			
<b>Week 5</b>			
<b>Week 6</b>			

Vienna, March 12, 2012

## Contents

<b>Preface</b>	<b>4</b>
General . . . . .	4
Acknowledgements . . . . .	4
<b>1 Hardware</b>	<b>6</b>
1.1 Boardcheck ★ . . . . .	6
1.2 Oscilloscope ★ . . . . .	6
1.3 Getting started ★ . . . . .	7
<b>2 Assembler</b>	<b>9</b>
2.1 General . . . . .	9
2.1.1 Demo program ★ . . . . .	9
2.1.2 Makefile ★ . . . . .	9
2.2 Digital I/O . . . . .	10
2.2.1 Logical operations ★ . . . . .	10
2.2.2 Input with floating pins ★ ★ . . . . .	12
2.2.3 LED rain ★ ★ . . . . .	13
2.2.4 Monoflop buttons ★ ★ . . . . .	14
2.2.5 Digital I/O ★ ★ . . . . .	16
2.2.6 LED rotator ★ ★ ★ . . . . .	17
2.2.7 Interrupt latency ★ ★ ★ . . . . .	18
2.2.8 LED curtain ★ ★ ★ . . . . .	18
2.3 Communication . . . . .	20
2.3.1 Simple UART ★ ★ ★ . . . . .	20
2.4 More advanced topics . . . . .	21
2.4.1 Using precompiled LCD module ★ . . . . .	21
2.4.2 Calling conventions – server ★ ★ . . . . .	22
2.4.3 Calling conventions – client ★ ★ ★ . . . . .	23
2.4.4 Fibonacci numbers ★ ★ ★ . . . . .	23
2.4.5 Arithmetic progression ★ ★ ★ . . . . .	24
2.4.6 Sieve of Eratosthenes ★ ★ ★ . . . . .	25
2.4.7 Using precompiled keypad module ★ . . . . .	27
<b>3 C</b>	<b>28</b>
3.1 General . . . . .	28
3.1.1 Demo program ★ . . . . .	28
3.1.2 Makefile ★ . . . . .	28
3.1.3 Floating point operations ★ . . . . .	28
3.2 Digital I/O . . . . .	30
3.2.1 Logical operations ★ . . . . .	30
3.2.2 Input with floating pins ★ ★ . . . . .	30
3.2.3 Voltage levels ★ ★ . . . . .	32
3.3 Interrupt . . . . .	32
3.3.1 Interrupt & callback demo ★ . . . . .	32
3.3.2 Interrupts ★ ★ . . . . .	33

3.3.3	Interrupt latency ★★	34
3.3.4	Interrupt modes ★★	34
3.4	Timer	36
3.4.1	Input capture ★★	36
3.4.2	PWM modes ★★	37
3.4.3	Generating periodic signals ★★	38
3.4.4	PWM signals and glitches ★★	39
3.5	Analog module	40
3.5.1	Analog conversion ★★	40
3.5.2	Noise ★★	41
3.5.3	Prescaler and accuracy ★★	42
3.6	Communication	43
3.6.1	UART receiver ★	43
3.6.2	UART sender ★	45
3.6.3	SPI ★★	45
3.6.4	I <sup>2</sup> C ★★	46
3.7	Architecture	47
3.7.1	Calling assembler functions from C ★★ ★	47
3.7.2	Watchdog ★★	48
3.7.3	Pipelining ★★	50
3.7.4	Memory considerations ★★	50
3.7.5	Dynamic memory analysis ★★	51
3.7.6	CPU usage ★★ ★	52
3.8	On-board hardware	53
3.8.1	Switches ★★	53
3.8.2	Button debouncing ★★	53
3.8.3	Using the precompiled LCD module ★	54
3.8.4	LCD ★★	55
3.8.5	Using the precompiled Graphic GLCD module ★★	56
3.8.6	Graphic LCD ★★	59
3.8.7	Touch panel ★★	59
3.9	External Hardware	60
3.9.1	Keypad ★★	60
3.9.2	Humidity & Temperature (SHT1X) ★★ ★	61
3.9.3	Light to frequency ★	61
3.9.4	MMC card ★★	62
3.9.5	RFID ★★ ★	63
3.10	Applications	64
3.10.1	Random number generator ★★	64
3.10.2	GPS 1PPS ★★	65
3.10.3	Alarm clock ★★ ★	65
3.10.4	Scrolling text ★★	66
3.10.5	Distance sensor ★★ ★	67
3.10.6	Correct LED dimming ★★ ★	68
3.10.7	Sinusoidal signal ★★ ★	69
3.10.8	RFID lock ★★ ★	69

3.10.9	Hygrometer ★★	70
3.10.10	Analog safe lock ★★	71
3.10.11	Light adaption ★★	71
3.10.12	Dew point ★★	72
3.10.13	Six's thermometer ★★	73
3.10.14	Bang-bang controller ★★	74
3.10.15	(Egg) timer ★★	75
3.10.16	Touchscreen calibration ★★	75
3.10.17	Graphical user interface ★★	76
3.11	Games	77
3.11.1	LED dice ★★	77
3.11.2	Simon says ★★	78
3.11.3	Human Response Time ★★	79

## Preface

### General

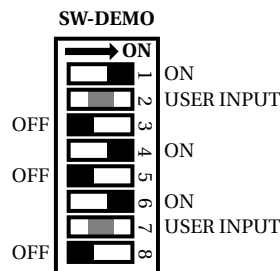
This set of exercises covers all course material, ranging from Assembler- (and C <sup>1</sup>)-Programming over in-depth treatment of the AVR microcontroller to the external hardware used. It has been designed to accommodate a wide range of experience levels, from novices to students with previous microcontroller experience.

To help you assess the difficulty of the exercises, we have rated each exercise with a difficulty level which is partitioned into 5 groups:

- ★ Very easy
- ★★ Easy
- ★★★ Moderate
- ★★★★ Advanced
- ★★★★★ Expert

Furthermore, if an exercise is based on knowledge conveyed in another exercise, this dependency is often stated as well to help you plan your schedule. For recommended exercises rated below moderate (★★★), we have put in references to the manuals and datasheets to help you get your bearings. The level of practical exams will be between easy (★★) and moderate (★★★). Applications will be at advanced (★★★★) level.

Pin configurations are shown with images like following, this demo-switch should only make clear where ON and OFF position of the switches are and which are used as user input.



Since you should have the freedom to customize the course to your needs, only some of the exercises, which cover the basic skills and knowledge necessary to pass the course, are recommended. From the remaining exercises, you may choose as you please.

Microcontroller programming and debugging can be very challenging and satisfying. Exploring unexpected behavior, formulating theories, testing them, and finally coming up with an explanation is both thrilling and extremely instructive. We hope that our exercises will trigger your interest and motivate you to explore relationships and interesting hardware behavior beyond the scope of our course, while at the same time conveying microcontroller programming skills and some basic hardware/electronics skills along the way.

We wish you a lot of fun!

---

<sup>1</sup>We expect you to be proficient in C. However, we offer some C exercises so you can refresh your knowledge.

## Acknowledgements

Our thanks go to the following people for contributing exercises to this collection<sup>2</sup>:  
Martin Biely, Thomas Hamböck, Thomas Mozgan, Christian Trödhandl

---

<sup>2</sup>If you have an idea for an application exercise that you believe is interesting and which can be done with our course hardware, mail your suggestion to the lecturer

# Exercises

## 1 Hardware

### 1.1 Boardcheck

★

#### Subjects

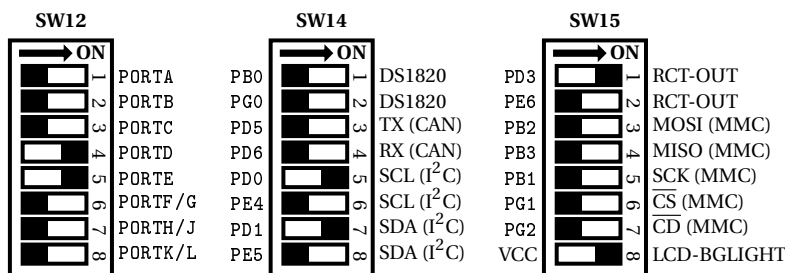
- Get in touch with the hardware

#### Task

Set up specified pin assignment and download the board test from [1] and flash it onto the board using `make install`. Now follow the instructions on the LCD and on the GLCD.

At first the frequency of the soldered crystal oscillator is measured using the real-time clock square wave output. The measured frequency should be about 16000000Hz. Then, after pressing any button on PORTB, the LEDs are tested. Turn the LEDs for all ports on. All LEDs should light up (instead of PORTA which switches between 2 patterns). After the LED test the analog-digital-converter is tested. Therefore you'll have to set **J15** to PF0. Then turn the trimmer to maximum and minimum afterwards. Then a rectangular signal is generated on PL3 and PL4. The frequency generated can be changed using the trimmer. You can choose out of 1Hz, 10Hz, 100Hz, 1kHz, 10kHz, 100kHz and 1MHz.

#### Pin assignment



Set **J18** to VCC (+) and **J12** to GND (⊥). Disconnect **J13** and **J15**. All other switches are set to off.

#### Remarks

- If you can't see anything on the LCD or the GLCD try to adjust the contrast!

### 1.2 Oscilloscope

★

Based on: [1.1](#)

## Subjects

- Get in touch with the hardware

## Task

Do everything described in [1.1](#) to get the signal generation running. Now connect the probes of the oscilloscope to PL3 and PL4 using an "EasyTest" module. Make sure you do not create short-circuits.

- Display the two signals on the oscilloscope.
- Experiment with the resolution (voltage and time).
- Find out how to freeze the picture.
- Shift the signal horizontally and vertically, zoom in and out of it.
- Measure the time (and frequency) between two edges of the signal from channel A. Use the cursors for this purpose. Do this with different frequency settings.
- Measure the duty cycles of both signals.
- Experiment with the trigger and find out how to change the trigger from channel A to channel B. Check out the trigger level.
- Let the oscilloscope trigger on edges (rising, falling).
- Configure the oscilloscope so that it detect peaks in the signal.
- Try the math mode and add the signal from PL4 to PL3. What can you see? Can you find out the PWM mode by looking at the added signal? If yes, which PWM mode is used? If not, why not?

The different PWM modes are described in ATmega1280 manual [\[2\]](#) on pages 150–156.

## Remarks

- Do not change settings you do not understand. If you manage to completely mess up the oscilloscope settings and do not know how to set it back to useful values, set it back to its factory defaults.
- If you are not sure the oscilloscope is set right, connect the probe to the output called "Tastkopfabgleich". It generates a square wave signal of 5V with 1kHz amplitude. Use this signal to get the oscilloscope settings right. If all else fails, set it back to its factory defaults.

## Pitfalls

- The probe has a slider that selects 1× or 10× attenuation. The slider should be at 1×. The oscilloscope has a corresponding setting, which should also be at 1×.

## 1.3 Getting started

★

### Subjects

- Set up the toolchain



### **Task**

Download the getting started tutorial from [\[3\]](#) and follow the instructions to get the toolchain running.

## 2 Assembler

### 2.1 General

#### 2.1.1 Demo program

★

##### Subjects

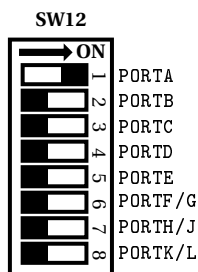
- Flashing the board
- Get in touch with assembler

##### Task

Take a look at the demo program you can download from the course homepage ([4, Assembler Demo]). The program contains one syntactical error. Call `make` to assemble the program, identify the location of the error from the assembler output, correct the error, and assemble again. Then download the resulting elf-file to the controller using `make install`. As soon as the download is finished, the controller starts executing the program.

Play around with the program to familiarize yourself with the assembly language. Make sure you know how to initialize the stack pointer, how to declare and use variables, how to set registers, how to jump to a label, how to do loops, how to call subroutines, and how to place code and data at specific memory locations.

##### Pin assignment



All other switches are set to off.

##### References

- 
- ATmega1280 manual [2], general purpose register file: p. 15
- ATmega1280 manual [2], stack pointer: p. 16
- ATmega1280 manual [2], SRAM data memory: p. 21
- AVR instruction set [5]

#### 2.1.2 Makefile

★

##### Subjects

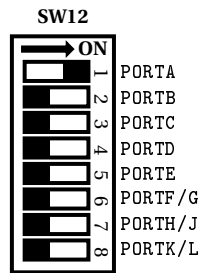
- Get in touch with the toolchain

## Task

Download the `demo_asm_make` from [6]. Get familiar with the toolchain and write a makefile that assembles and links the demo program and downloads it to the microcontroller. Do NOT use the one provided by the assembler demo program.

Play around with the makefile to familiarize yourself with the toolchain and the options to the assembler and the linker.

## Pin assignment



All other switches are set to off.

## Questions

1. Our microcontroller has 3 different memories: Flash, EEPROM and SRAM. Which memory can be programmed how (during program execution / at program download)?

## Hints

- Assembler: `avr-as`.
- Linker: `avr-ld`
- Programmer: `avrprog2`

## References

- 
- ATmega1280 manual [2], general purpose register file: p. 15
- ATmega1280 manual [2], stack pointer: p. 16
- ATmega1280 manual [2], SRAM data memory: p. 21
- AVR instruction set [5]

## 2.2 Digital I/O

### 2.2.1 Logical operations

★

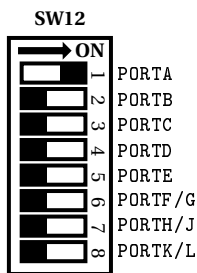
## Subjects

- Get a slightly deeper sight into assembler
- Basic I/O

**Task**

Set PA3:0 to input with internal pull-up and PA7:4 to output and use PORTA7:4 for LED0-3. In your program, implement the following operations and display their results on the LEDs:

- $LED0 := (PA_1 \wedge PA_2) \vee PA_3$
- $LED1 := (PA_1 \vee \neg PA_2) \wedge PA_3$
- $LED2 := PA_1 \oplus PA_2$  [Antivalence]
- $LED3 := PA_1 \Leftrightarrow PA_2$  [Equivalence]

**Pin assignment**

All other switches are set to off.

**Remarks**

- $PA_n$  denotes bit  $n$  of port PORTA. Bit numbering starts with 0.
- In which ways can you extract only one bit from a register?  
Whats the fastest method for  $Rx_0$ ,  $Rx_1$ ,  $Rx_4$  and  $Rx_7$  if 1) the value of  $Rx$  isn't necessary afterwards and 2) the value shouldn't be destroyed?

**Questions**

1. Do you need to set the stack pointer? Why or why not?
2. What is the reset value of the SP? When the SP is needed, why should one set it initially to the end of the SRAM?
3. What is the first address of the data memory that is available for the user program? What is located before that address?
4. There are two different types of set/clear bit operations: Those that operate on registers in general (SBR/CBR), and those that operate on I/O registers (SBI/CBI). Explain the differences between the two types.
5. Explain the differences between NEG and COM. If you want to invert a single bit in a register, which of the instructions NEG, COM, EOR can you use? (Show how; multiple answers possible.)

**Pitfalls**

- Note that the bit set and bit clear instructions work only on the first 32 I/O registers, see AVR instruction set [\[5\]](#).

- When using `.equ`, do not use names starting with `{r|R}`. The Assembler expects a register number after the `r`.

## 2.2.2 Input with floating pins

★ ★

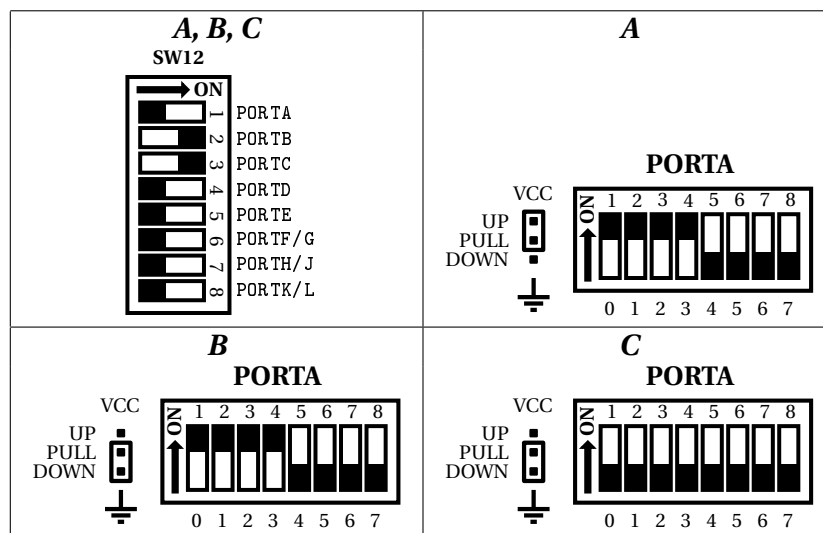
### Subjects

- Basics of I/O

### Task

Set pin PA0-7 to input and PB0-7, PC0-7 to output and do *not* activate the pull-up. In the main loop, display register PORTA on PB0-7 and bit PINA on PC0-7.

### Pin assignment



All other switches are set to off.

### Questions

1. Set up pin assignment **A** and touch the pin header of PORTA on the right with your fingers. Then also press the buttons for PA0-7. What do you see on the LEDs? What can you expect to see?
2. Set up pin assignment **B** and repeat the actions of the previous question. What do you see now on the LEDs? What can you expect to see?
3. For completeness' sake set up pin assignment **C** and activate the internal pull-up for PORTA0-3 and repeat the experiment.
4. Explain the difference between the PORT and PIN registers when a pin is set to input.

### Pitfalls

- When you touch the pin header be aware that you don't touch the pull-up/pull-down pins else you maybe won't see anything.

- When you have no pull-up/down you should set the buttons to high-active (set **J12** to VCC (+))
- When you set the external or internal pull-up you should set the buttons to high-active (set **J12** to GND ( $\perp$ ))
- When you set the external pull-down you should set the buttons to low-active (set **J12** to VCC (+))

## References

- ATmega1280 manual [2], I/O ports: p. 70–104, registers PORTx, DDRx, PINx

### 2.2.3 LED rain

★ ★

#### Subjects

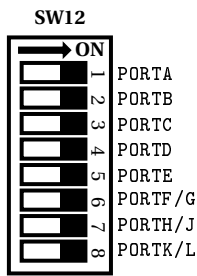
- Basics of I/O
- Get in touch with more instructions

#### Task

Set PORTA-G to output and update with a delay (you can use a simple busy-wait loop) the Ports with given formula. Use two 8 bit counters, counter1 increments by one per update and counter two decrements by  $-3$  per update.

$CNT_1$  ... counter 1  
 $CNT_2$  ... counter 2  
 $PORTA := CNT_1 \oplus CNT_2$   
 $PORTB := (\neg CNT_1) + CNT_2$   
 $PORTC := CNT_1 - CNT_2$   
 $PORTD := CNT_1 \wedge (-CNT_2)$   
 $PORTE := (CNT_1 * CNT_2) \wedge 0xFF$   
 $PORTF := (CNT_1 \gg 1) \vee CNT_2$   
 $PORTG := (SW(CNT_1) \wedge 0x0F) \vee (CNT_2 \wedge 0xF0)$   
 $\neg$  ... bit-wise inversion  
 $-$  ... negation (2s complement)  
 $\gg$  ... right shift (logical)  
 $SW$  ... swap operation

### Pin assignment



All other switches are set to off.

### Hints

- For the counters choose two registers in the scratchpad.
- Useful commands may be: ADD, AND, ANDI, COM, EOR, MUL, NEG, OR, ROL, SUB, SWAP

### Questions

1. Why have we used  $(SW(x) \wedge 0x0F)$  instead of  $(x \gg 4)$ ? What are the advantages/disadvantages?
2. What are the differences between AND and ANDI, OR, ORI?
3. We want to increment the register R16 by three, does

```
1 ldi R17, 3
2 add R16, R17
```

the same as

```
1 subi R16, -3
```

does? Works this trick also for numbers bigger than 128 (smaller than  $-128$ )?

### References

- ATmega1280 manual [2], I/O ports: p. 70–104, registers PORTx, DDRx, PINx

#### 2.2.4 Monoflop buttons

★ ★

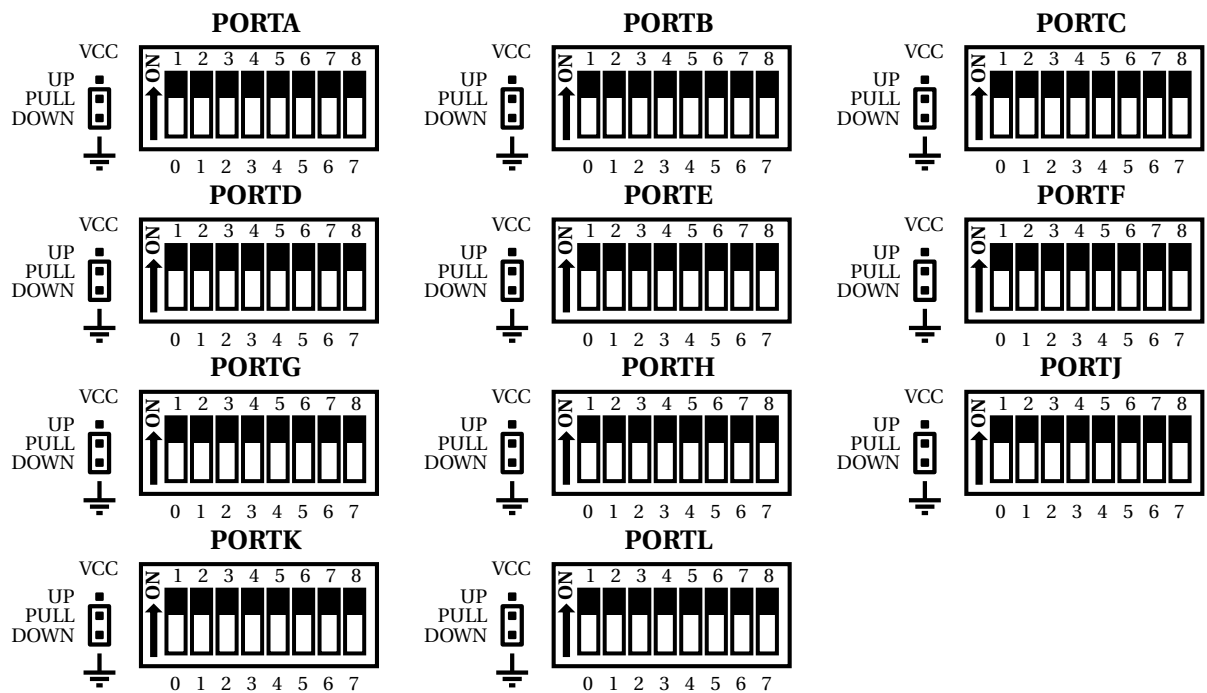
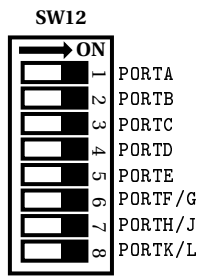
### Subjects

- Basics of I/O

### Task

Set P{A-L} to input and set J12 to VCC (+). In the main loop read in PIN{A-L} and set the PORT{A-L} registers to those values and also set the DDR accordingly (if the output of a pin is low it should be input, if it is high it should be an output). ( $PORTx := PINx$  and  $DDRx := PORTx$ ). If all pins are set to high reset all pins to input and low and start over.

## Pin assignment



All other switches are set to off.

## Questions

1. Describe the effect when you press the buttons.

## Hints

- The CPSE instruction can be very useful.
- Set the DDR before the PORTs to avoid driver conflicts.
- There exists no PORTI.
- PORT{H|J|K|L} can't be accessed direct using IN/OUT, use LDS/STS instead.

## Pitfalls

- Don't forget to set the buttons to high-active (J12 to VCC (+)).



## References

- ATmega1280 manual [2], I/O ports: p. 70–104, registers PORTx, DDRx, PINx

### 2.2.5 Digital I/O

★ ★

## Subjects

- Simple User I/O

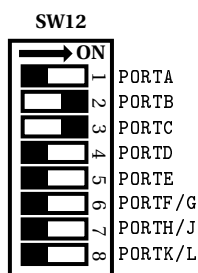
## Task

Set PA3 to output. In the initialization part of your program, put in the following code:

```
1  sbi  PORTA , 3      ; set PA3 high
2  in   Ra , PINA      ; read PINA
3  in   Rb , PINA
4  in   Rc , PINA
```

Display the Ra on PORTB, Rb on PORTC and Rc on PORTD. Ra-Rc can be any suitable free registers. The main loop of the program should be empty.

## Pin assignment



All other switches are set to off.

## Questions

1. What do you see on the LEDs and why?
2. Judging from the output, what is a lower bound on the input delay? What is an upper bound? Can you give tight bounds just by observing your program's output? Justify your answers.
3. What do your observations imply for your programming practice?

## Pitfalls

- When using `.equ`, do not use names starting with `{r|R}`. The assembler expects a register number after the `r`.

## References

- ATmega1280 manual [2], input delay: p. 72–73

### 2.2.6 LED rotator

\*\*\*

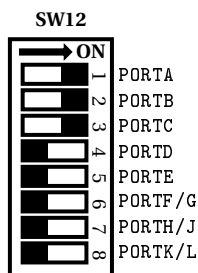
#### Subjects

- Basics of I/O
- External interrupt

#### Task

We use  $LED_n$  for the output of  $PA_n$ ,  $PB_n$  and  $PC_n$ . Initially turn on LED0. Write an interrupt service routine (ISR) that rotates LED3:0 to the left by one whenever it is called (that is, if LED $i$  is on prior to calling the ISR, then after calling the ISR LED $\{i + 1 \bmod 4\}$  should be on. Install this ISR as the INT0 ISR and let the LEDs rotate by one whenever the button is pressed down. Furthermore install an similar method but using the formula LED $\{i - 1 \bmod 4\}$  for INT1 ISR. Set the Buttons to pull down (J12 to GND ( $\perp$ )) and use falling edge for INT0.

#### Pin assignment



All other switches are set to off.

#### Questions

1. Explain the difference between the RET and the RETI instructions. Why do you need RETI to return from an ISR? What happens if you use RET instead?
2. What happens if you are in an ISR and an interrupt with higher priority occurs? Will your ISR get interrupted (assume that you do not touch the Global Interrupt Enable bit in the ISR)?
3. Some experienced programmers save the status register on the stack in the ISR, regardless of what the ISR does. What reasons could they have?

#### Pitfalls

- Note that the addresses given in the interrupt vector table of the ATmega1280 manual [2] (and defined in `m1280def.inc`) are *word* addresses, whereas the Assembler requires byte addresses. So you will need to multiply them by 2.

#### Hints

- To get started you can have a look at the assembler interrupt demo [7].
- The assembler instruction SWAP might be helpful. Also, the half-carry bit H might come in handy.

- Like with most controllers, the port pins of the ATmega1280 have alternate functions. You just have to enable the alternate function (in this case, the external interrupt function) to be able to use it.
- After the reset, interrupts are globally disabled. Use the SEI/CLI instructions to globally enable/disable interrupts.

### References

- ATmega1280 manual [2], reset and interrupts: p. 18–20
- ATmega1280 manual [2], interrupt vectors: p. 105–106
- ATmega1280 manual [2], external interrupts: p. 112–117
- ATmega1280 manual [2], instruction set: p. 336–338, instructions SEI, CLI, RETI

### 2.2.7 Interrupt latency

★ ★ ★

#### Subjects

- Basics of I/O
- External interrupt

#### Task

Initialize LED0 for output and turn it off. Initialize INT0 as output and set it to 0. Turn on the external interrupt feature of INT0 and let an interrupt be generated for the rising edge. Then set INT0 to 1. In the INT0 ISR, turn on LED0 in the first instruction. Use the oscilloscope to determine the interrupt latency from the signals on INT0 and LED0.

#### Questions

1. Which delays make up the interrupt latency of the AVR controller?
2. What additional delays (apart from the latency itself), if any, are part of the time you have measured?
3. What determines the accuracy of your measurement? How accurate can you get?

#### Hints

- For measuring with the oscilloscope you can use the "EasyTest" modules and jumper wires to prevent wrong results or short circuits.

### 2.2.8 LED curtain

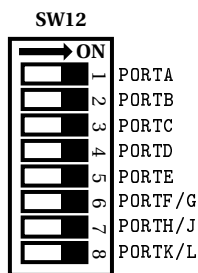
★ ★ ★

#### Subjects

- Basics of I/O
- Indirect memory addressing
- Timer interrupt

**Task**

Set `PORT{A-G}` to output and set a timer (e.g., Timer 0) to CTC mode with a frequency of  $50\text{Hz}$  (Duration of  $20\text{ms}$ ). For this timer you can use a prescaler of 256 and an `OCR` value of 1250. Whenever the timer interrupt occurs shift `PORT $x$` ,  $x \in \{A - G\}$  one place left and add one (start with  $x = A$ ). Then select the next port for  $x$ . If `PORTA` equals to `0xFF` switch to shift left without adding one, and if `PORTA` equals to `0x00` switch back to shift left and add one.

**Pin assignment**

All other switches are set to off.

**Hints**

- The commands `adiw` or `ldd/std` may be useful.
- For loading the `OCR $x$ H` use `hi8( $n$ )` and for `OCR $x$ L` use `lo8( $n$ )`.

**Questions**

1. After solving this exercise extend your program that it includes `PORT{H-L}`.

**Pitfalls**

- `PORT{H|J|K|L}` have a different I/O address.
- For accessing `PORT{A-G}` with `LDS/STS/LD/ST` you will have to add `+0x20` to the port address.
- `PORT{H|J|K|L}` can't be accessed direct using `IN/OUT`, use `LDS/STS` instead.

**References**

- AVR instruction set [5]
- ATmega1280 manual [2], timer CTC mode description: p. 149–150
- ATmega1280 manual [2], timer register descriptions: p. 158–168
- ATmega1280 manual [2], interrupt vectors: p. 105–106

## 2.3 Communication

### 2.3.1 Simple UART

★ ★ ★

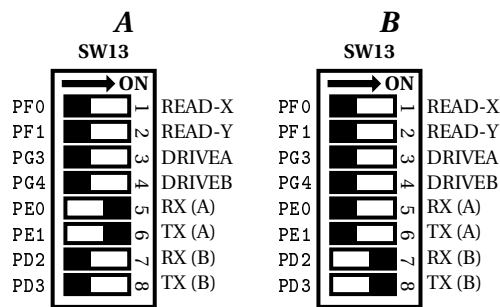
#### Subjects

- Using the UART
- UART interrupt

#### Task

Enable the UART using the asynchronous mode, data format 8N1 (8bit, no parity and one stop bit) and a baud-rate of 9600Baud (don't enable the double speed mode). Enable the receiver (RXEN), the transmitter (TXEN) and the receive interrupt (RXCIE). In the receive interrupt read the UDR register into a temporary register, increment the temporary register by one and write it again out to UDR. If you send 'A' to the MCU, 'B' should be sent to the PC ('A' has ASCII code 65 and 'B' is 66).

#### Pin assignment



All other switches are set to off.

#### Remarks

- Use pin assignment **A** for UART0 and RS-232A (PE0/1).
- Use pin assignment **B** for UART1 and RS-232B (PD2/3).

#### Hints

- Choose the baud-rate from table 22-9 on page 227–231 of the ATmega1280 manual [2].
- Attach the USB to serial converter and connect it to RS-232A or B (as you chose the pin assignment) and use minicom, cutecom or gtkterm for UART communication.

#### Pitfalls

- Don't enable the data register entry interrupt and the transmit interrupt.

## References

- ATmega1280 manual [2], precalculated UART baud-rate: p. 227–231
- ATmega1280 manual [2], UART: p. 205–240

## 2.4 More advanced topics

### 2.4.1 Using precompiled LCD module ★

#### Subjects

- Simple user I/O

#### Task

Download the LCD example from the course web page [8] and flash it onto the MCU (make install). Look at the demo.s and get familiar with the precompiled assembler LCD driver.

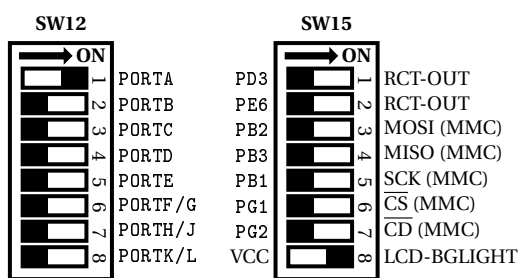
If you want to write your own program using the precompiled LCD module you will need following register definitions. As you see, as calling conventions, the registers 24–26 were chosen.

```
1 .equ    param1, 24
2 .equ    param2, 25
3 .equ    param3, 26
```

Further there exists no routine for displaying 8bit numbers on the LCD, use the 16bit routine with high-register 0.

The  $x$  axis goes to the right and has a length of 16Characters (0–15). The  $y$  axes is the line (0 = first line, 1 = second line).

#### Pin assignment



All other switches are set to off.

#### Provided functions

- Initialize LCD

```
1 call    initLCD
```
- Clear screen

```
1 call    clearScreen
```
- Display single character on LCD

```
1 ldi    param1, 0 | (0<<7) ; x | (y<<7) position
2 ldi    param2, 'X'        ; character to display
3 call   dispChar           ; display character
```

- Display 16bit number on LCD

```
1 ldi    param1, 0 | (0<<7) ; x | (y<<7) position
2 ldi    param2, lo8(512)    ; lo(value) to display
3 ldi    param3, hi8(512)    ; hi(value) to display
4 call   dispUint16         ; display number
```

- Display string in flash memory on LCD

```
1 ldi    param1, 0 | (0<<7) ; x | (y<<7) position
2 ldi    ZL, lo8(String)     ; lower PMEM address of string
3 ldi    ZH, hi8(String)     ; higher PMEM address of string
4 call   dispStrPMEM        ; display string
```

- Display string in SRAM memory on LCD

```
1 ldi    param1, 0 | (0<<7) ; x | (y<<7) position
2 ldi    ZL, lo8(String)     ; lower SRAM address of string
3 ldi    ZH, hi8(String)     ; higher SRAM address of string
4 call   dispStrSRAM        ; display string
```

## 2.4.2 Calling conventions – server

★★

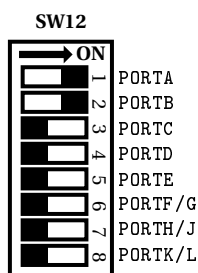
### Subjects

- Calling conventions

### Task

Download the calling conventions server example from the course web page [9] and complete the file `main.s`. The function `incrementU16` takes an unsigned 16bit number as parameter (on the stack), increments it by one and gives the result back on the stack. The high value of the 16bit value is pushed first and then the low value is pushed, so the low value is on top of the stack and the high value on the second position. See `increment.s` for more insight. The main function should call the `incrementU16` function to increment a local 16bit counter (keep the counter in some GP-register) and output the current value to `PORTA:B`. The low value of the counter should be output to `PORTA` and the high value to `PORTB`.

### Pin assignment



All other switches are set to off.

### 2.4.3 Calling conventions – client

\*\*\*

#### Subjects

- Calling conventions

#### Task

Download the calling conventions client example from the course web page [10] and complete the file `increment.s`. The function `incrementU16` takes an unsigned 16bit number as parameter, increments it by one and gives the result back. The parameters are pushed onto the stack in the `main.s`. At first the high value and then the low value is pushed onto the stack. Then the `incrementU16` function is called. Afterwards the result is on the stack again, at first low value and then the high value.

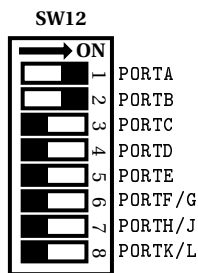
To handle such function calls right you'll have to

1. Push all touched registers onto the stack.
2. Load the stack pointer (SPL, SPH) into a memory pointer (X, Y or Z).
3. Add  $n + k + 2$  to the memory pointer where  $n$  is the number of registers saved in 1 and  $k$  is the number of parameters passed by. 2 is the size of the program counter.

As long as this number is  $\leq 63$  you can use `ldd / std` or `adiw` for this task.

4. Load the parameters using `ld / ldd` with the chosen memory pointer.
5. Calculate the result and save it back using `st / std`.
6. Restore the saved values from 1

#### Pin assignment



All other switches are set to off.

#### Remarks

- For this type of calling conventions the number of return values must be the same as the number of parameters.

### 2.4.4 Fibonacci numbers

\*\*\*

#### Subjects

- Simple user I/O
- External interrupt
- Use precompiled LCD module

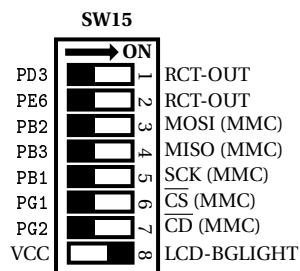


### Task

Download the LCD example from the course web page [8] and use the delivered precompiled LCD module (lcd.o). Use *16bit* integer arithmetic to calculate the  $n$ -th Fibonacci number  $F(n)$ . After calculating the number, display the number on the LCD screen using the precompiled LCD module. You can display  $n$  in the first line ( $y = 0$ ) and  $F(n)$  in the second line ( $y = 1$ ).

Start with the first one, displaying  $F(0) = 0$ . Use the external interrupt (e.g., INT7=PE7) to increment  $n$  and display the next  $F(n)$  on the LCD.

### Pin assignment



All other switches are set to off.

### Questions

1. What is the highest  $n$  for which the Fibonacci number can be stored in a *16bit* integer?
2. What is the highest  $n$  for which the Fibonacci number could be stored in a *32bit* integer?
3. If you do a recursive implementation without optimizations, how much stack do you need to compute  $F(n)$ ? Give a formula for the stack usage  $S(n)$  (with proof).
4. How much time  $T(n)$  does your implementation take to compute  $F(n)$ ? Give an estimation of  $T(n)$  (upper and lower bound) in both clock cycles and seconds.

### Hints

- The Fibonacci numbers are defined by the formula  $F(n) = F(n-1) + F(n-2)$ , where  $F(0) = 0$  and  $F(1) = 1$ .
- To compute  $S(n)$ , first write down the stack usage for  $n = 0, 1, 2, \dots, 5$ . You will see a certain pattern and a correlation with  $F(n)$  from which you can derive a formula. Prove this formula by induction.

### 2.4.5 Arithmetic progression

\*\*\*

### Subjects

- Simple user I/O
- External interrupt
- Use precompiled LCD module

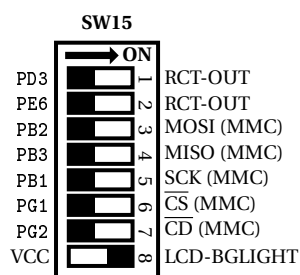
**Task**

Realize following simple arithmetic sequence:

$$f(n) = \begin{cases} \frac{n}{2} & \text{if } n \text{ is even} \\ 3n + 1 & \text{otherwise} \end{cases}$$

(This exercise is inspired by [11]).

Start with a given  $n$  and display it on the LCD. Activate the external interrupt (e.g., INT7). Whenever the external interrupt occurs calculate the next number of the sequence and display it with its position in the sequence on the LCD.

**Pin assignment**

All other switches are set to off.

**Questions**

1. What's the fastest way implement the multiplication  $3n$  ?  
What are the alternatives and why is yours faster?

**Remarks**

- Define the starting number of the sequence  $n$  in only one place (with . equ), so that it can be changed easily.

**2.4.6 Sieve of Eratosthenes**

★ ★ ★

**Subjects**

- Simple user I/O
- External interrupt
- Realize a simple algorithm
- Indirect memory access
- Use precompiled LCD module

**Task**

Program the Sieve of Eratosthenes (see [12]) for prime numbers between 2 and TOP.

The algorithm works like this: Mark all numbers between 2 and TOP. Select the first marked number (2) and clear your marker from all multiples of this number (but not

from the number itself). Select the next marked number (3) and again clear all its multiples. Repeat this (5, 7, 11, ...) until there are no more remaining marked numbers. All numbers still marked are prime numbers.

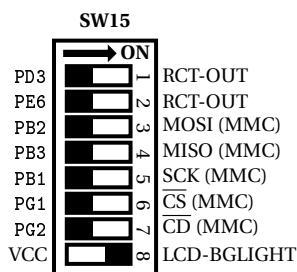
At first start the algorithm and determine all wanted primes, then display the first (2) on the LCD and enter IDLE sleep mode. Whenever the external interrupt (e.g., INT7) occurs display the next prime on the LCD.

Set TOP to the maximum number of primes that can be determined.

### Remarks

- Define TOP in only one place (with .equ), so that it can be changed easily.
- Use subroutines for deleting multiples and for counting the remaining prime numbers.

### Pin assignment



All other switches are set to off.

### Questions

1. Let us assume that you have initially set the SP to the end of the RAM. If you push the contents of R0 on the stack after it has been initialized, then at which address (the actual address, no symbols please) is the data located and to which address does the SP point?
2. What would happen if the program did not initialize the SP? Could you overwrite code? Could you overwrite data?
3. Can you easily state the time complexity of the algorithm with respect to deleting multiples (assume that the array size is  $n$ )? If so, do it. If not, explain why.

### Pitfalls

- 1 is not a prime number. Start with 2.

### Hints

- Use an array (some consecutive and unused portion of the SRAM) to store the markers, and use one byte per number to make things easier.
- Use indirect addressing with post-increment to iterate through the array.

## 2.4.7 Using precompiled keypad module

★

### Subjects

- Simple user I/O

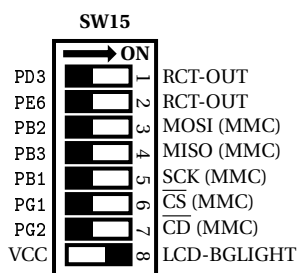
### Task

Attach the 4x4 keypad module to PORTA and download the keypad example from the course web page [\[13\]](#) and flash it onto the MCU (make install). Look at the demo.s and get familiar with the precompiled assembler keypad driver.

If you want to write your own program using the precompiled keypad module you will need following register definition. As you see, as calling conventions, the register 24 was chosen (the same as the LCD module uses).

```
1 .equ param1, 24
```

### Pin assignment



All other switches are set to off.

### Provided functions

- Initialize Keypad

```
1 call initKeypad
```

- Wait for key

```
1 call getKey
2 ; key value is now in param1
```

- Convert key pressed to according value

```
1 ; param1 should contain the key value
2 call keyToNum
3 ; now in param1 the value of key pressed is contained
```

- Convert key pressed to according character

```
1 ; param1 should contain the key value
2 call keyToChar
3 ; now in param1 the character of key pressed is contained
```

## 3 C

### 3.1 General

#### 3.1.1 Demo program

★

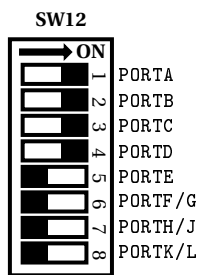
##### Subjects

- Flashing the board
- Get in touch with avr-gcc

##### Task

Download the C demo program from [14]. use make to compile the program and make install to flash the demo file onto the microcontroller.

##### Pin assignment



All other switches are set to off.

#### 3.1.2 Makefile

★

##### Subjects

- Get in touch with the toolchain

##### Task

Use the demo .c file from 3.3.1 and write your own Makefile for it.  
Use avr-gcc for compiling and linking.

#### 3.1.3 Floating point operations

★

##### Subjects

- Usage of floating points
- Execution time of floating point operations

##### Task

Globally define the following variables:

```
1 volatile uint16_t erg;
2 float      fA = 0.8;
3 float      fB = 0.5;
4 uint16_t    A = 8;
5 uint16_t    B = 5;
```

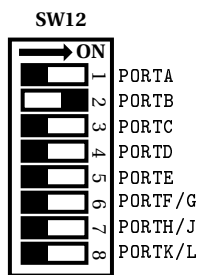
Initialize PB0 as output and set it to zero. Set PA0 to input and activate the pull-up. In your program, set PB0 to 1, then check the value of PA0 and execute `erg = (uint16_t)(10*fA + fB)` if PA0 is ON and `erg = (10*A + B)/10` if PA0 is OFF. After that, set PB0 back to 0 and enter an empty endless loop.

To set PA0 to ON or OFF use the button attached to PA0.

As is apparent, both variants compute the same result, which is 8 (the +0.5 resp. +5 are for rounding to the nearest integer). The first uses floating point operations, the second computes the same result, but converts the floats to integers (by multiplication with 10 to do away with the comma), does everything in integer arithmetic, and then (integer) divides the result by 10 to revert the previous multiplication.

Connect the oscilloscope to PB0 and measure the duration of the two calls (reset the controller to restart with a different PA0 setting).

### Pin assignment



All other switches are set to off.

### Questions

1. How long is the execution time of each of the functions (you can ignore the constant time that is added by checking the switch and calling the function)?
2. Compare the code sizes of the two functions (check the list file). What is the difference in code size (in bytes)?
3. What conclusions can you draw from your previous answers?
4. As an experiment, put the keyword `const` before the declarations of `fA` and `fB`. Measure the execution times again, and take a look at the list file. Describe your observations. What conclusions do you draw?

### Pitfalls

- Make sure the variable definitions are global, that is, outside of `main()`, otherwise you won't see any difference.
- When using the buttons with internal pull-up check that **J12** is set to GND ( $\perp$ ).

## 3.2 Digital I/O

### 3.2.1 Logical operations

★

#### Subjects

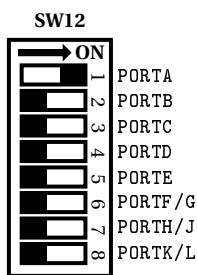
- Basic I/O

#### Task

Set PA0-3 to input with internal pull-up and PA4-7 to output and use PORTA4-7 for LED0-3. In your program, implement the following operations and display their results on the LEDs:

- $LED0 := (PA_1 \wedge PA_2) \vee PA_3$
- $LED1 := (PA_1 \vee \neg PA_2) \wedge PA_3$
- $LED2 := PA_1 \oplus PA_2$  [Antivalence]
- $LED3 := PA_1 \Leftrightarrow PA_2$  [Equivalence]

#### Pin assignment



All other switches are set to off.

#### Remarks

- $PA_n$  denotes bit  $n$  of port PORTA. Bit numbering starts with 0.

#### Questions

1. If you did the corresponding assembler exercise, now compile your program with `avr-gcc -S` and compare resulting `.S` file with your assembler solution. Which do you like more? Which is more efficient?

### 3.2.2 Input with floating pins

★★

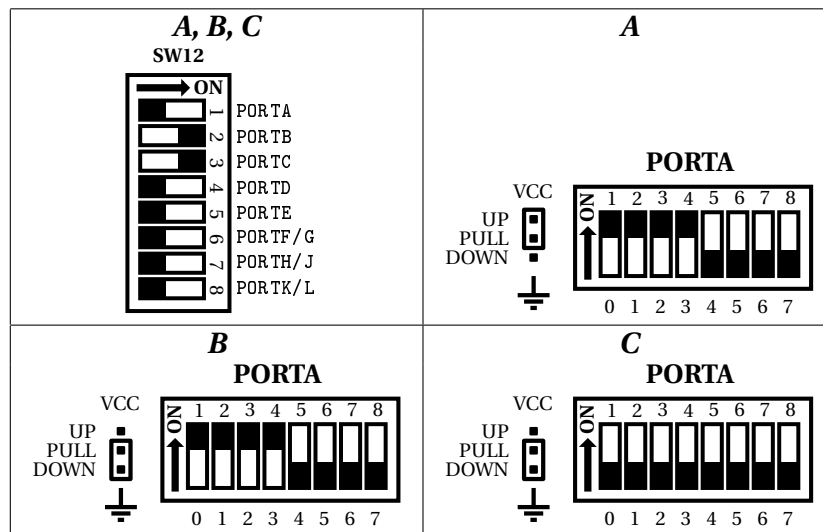
#### Subjects

- Basics of I/O

#### Task

Set pin PA0-7 to input and PB0-7, PC0-7 to output and do *not* activate the pull-up. In the main loop, display register PORTA on PB0-7 and bit PINA on PC0-7.

### Pin assignment



All other switches are set to off.

### Questions

1. Set up pin assignment **A** and touch the pin header of PORTA on the right with you fingers. Then also press the buttons for PA0-7. What do you see on the LEDs? What can you expect to see?
2. Set up pin assignment **B** and repeat the actions of the previous question. What do you see now on the LEDs? What can you expect to see?
3. For completeness' sake set up pin assignment **C** and activate the internal pull-up for PORTA0-3 and repeat the experiment.
4. Explain the difference between the PORT and PIN registers when a pin is set to input.
5. If you did the corresponding assembler exercise, now compile your program with `avr-gcc -S` and compare resulting `.S` file with your assembler solution. Which do you like more? Which is more efficient?

### Pitfalls

- When you touch the pin header be aware that you don't touch the pull-up/pull-down pins else you maybe won't see anything.
- When you have no pull-up/down you should set the buttons to high-active (set **J12** to VCC (+))
- When you set the external or internal pull-up you should set the buttons to high-active (set **J12** to GND ( $\perp$ ))
- When you set the external pull-down you should set the buttons to low-active (set **J12** to VCC (+))



### 3.2.3 Voltage levels

★ ★

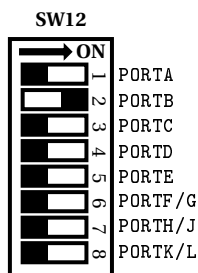
#### Subjects

- Advanced I/O

#### Task

Set **J15** to PF0. Set PA0 to digital input and continuously display its state on PB0. Connect the oscilloscope to PF0 and PB0.

#### Pin assignment



All other switches are set to off.

#### Questions

1. Which voltage levels map to a logical "1", which map to a "0"? When does the controller switch from one logic state to the other?
2. What are the theoretical bounds for the voltage levels as stated by the manual? Do they match your observations?
3. If you set the trimmer (PF0) to a voltage level between the upper bound for "0" and the lower bound for "1", does this increase the probability for a meta-stable state?

## 3.3 Interrupt

### 3.3.1 Interrupt & callback demo

★

#### Subjects

- Basic I/O
- External interrupt

#### Task

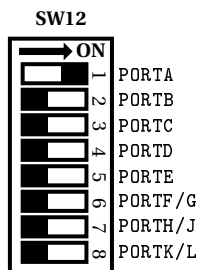
Download the C interrupt & callback demo from [15] and get known to interrupts and function pointers (mostly used as callback) in C.

The demo file uses a timer with a period of 250ms. Whenever the timer ISR gets called the function specified by the function pointer gets called with a value. Whenever PE7 (INT7) or PE6 (INT6) is called the first time in current periodic the function is changed

from increment to decrement and vice versa. The function is only changed once per period to reduce bouncing.

The timer 1 interrupt is blocking, INT7 is non-blocking and INT6 is an alias for INT7.

### Pin assignment



Set **J12** to GND ( $\perp$ ). All other switches are set to off.

### 3.3.2 Interrupts

★ ★

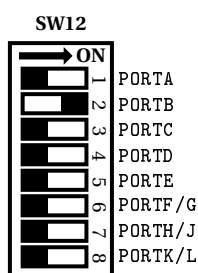
#### Subjects

- Basic I/O
- External interrupt

#### Task

Setup PORTB3:0 as output. Initially turn on PB0. Write an interrupt service routine (ISR) that rotates PB3:0 to the left by one whenever it is called (that is, if  $PB_i$  is on prior to calling the ISR, then after calling the ISR  $PB_{\{i+1 \bmod 4\}}$  should be on. Install this ISR as the INT0 ISR and let the LEDs rotate by one whenever the button is pressed down.

### Pin assignment



All other switches are set to off.

#### Pitfalls

- Don't forget to enable the global interrupts.

#### Hints

- To get started you can have a look at the assembler interrupt demo [\[16\]](#).

### 3.3.3 Interrupt latency

★ ★

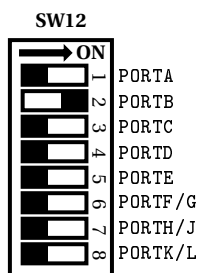
#### Subjects

- External interrupt

#### Task

Initialize PB0 for output and turn it off. Initialize INT0 as output and set it to 0. Turn on the external interrupt feature of INT0 and let an interrupt be generated for the rising edge. Then set INT0 to 1. In the INT0 ISR, turn on PB0 in the first instruction. Use the oscilloscope to determine the interrupt latency from the signals on INT0 and PB0.

#### Pin assignment



All other switches are set to off.

#### Questions

1. Which delays make up the interrupt latency of the AVR controller?
2. What additional delays (apart from the latency itself), if any, are part of the time you have measured?
3. What determines the accuracy of your measurement? How accurate can you get?

### 3.3.4 Interrupt modes

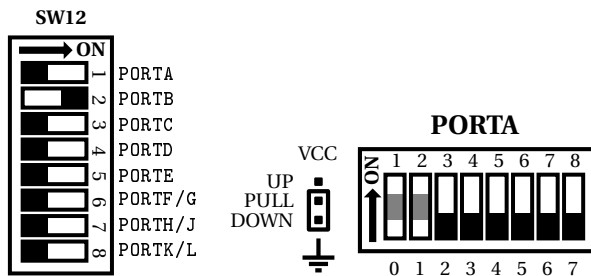
★ ★

Based on: [3.8.1](#)

#### Subjects

- Switches
- External interrupt

### Pin assignment



All other switches are set to off.

### Task

Configure INT0 as an input and install an ISR for it. Take the interrupt mode configuration of the MCUCR from PA1 : 0 (if PA1 : 0 are both ON, that is, they both read 0, then the interrupt mode bits should be set to 0). In the INT0 ISR, rotate the LEDs (PORTB7 : 0) by one (initially, PB0 should be turned on). The ISR should react when the according button is pressed down. To realize the switches on PA1 : 0 set DDRA1 : 0 to input and PORTA1 : 0 to pull-up. You can enable PA1 : 0 using the external pull-down (switches are low-active).

### Remarks

- You may put a delay into your ISR for observation purposes (especially for observing the effect of the level interrupt). But please be aware that busy-wait delays in ISRs are a bad thing and should be avoided.

### Questions

1. For each of the interrupt modes, describe what happens when you press/release the button and while it is pressed.
2. If you set the interrupt mode to low level and press the button for one second, how often is the ISR called?
3. For each interrupt mode, state which sleep modes it can interrupt.

### Pitfalls

- Do you need the pull-ups for the button?
- Don't enable the LEDs for PORTA, otherwise the pull-down switches won't work!

### Hints

- You can compute the answer to Question 2 from the information in the ATmega1280 manual [2]. But if you cannot figure out how to do it, you may also determine the value experimentally.

## 3.4 Timer

### 3.4.1 Input capture

★ ★

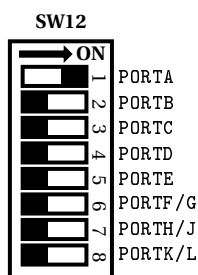
#### Subjects

- Input capture

#### Task

Set the ICP1 pin and PORTA to output and PD4 to input. We want to measure the time between button presses. To this aim, activate the timer with prescaler 1024 and just let it run freely. Install an ISR (blocking) for PD4 which triggers an input capture interrupt on ICP1. In the ICP ISR, read the captured timestamp and compute the time since the last capture interrupt. Display the time (in seconds, with one place behind the comma) since the last capture on the LEDs. Use a BCD representation and display the seconds (0 – 9) on PA3 : 0, and the fractional part (0 – 9) on PA7 : 4.

#### Pin assignment



All other switches are set to off.

#### Remarks

- This exercise is to show you both the input capture feature and the fact that input captures can be triggered by software by setting the ICP1 pin. Therefore, pin ICP1 is not connected to any external device, and its signal is controlled by software.
- You can only display values from 0.0 to 9.9 seconds. To determine the value you should display, use round-to-nearest. In case of an overflow, turn on all LEDs (PB7 : 0).
- Do not forget that the timer can wrap more than once between two button presses. You need to count those wrap-arounds.
- Note: Normally, it would be a better solution to directly use the button attached to ICP1. But in this exercise, we took the opportunity to not only show you how input capture works, but also how an input capture event can be triggered by software.

#### Questions

1. Would it have been sufficient to simply read the current count register in the INT0 ISR instead of triggering an input capture interrupt?

2. How close together can two interrupts occur and still be distinguished (an estimation suffices)?
3. Which events can trigger an input capture interrupt?

### Pitfalls

- Do not forget to set ICP1 to output.

### Hints

- To get the number you want to display, compute the time  $T$  between two capture interrupts in a unit of  $100ms$ . Display  $T/10$  on PB3 : 0, and  $T\%10$  on PB7 : 4.

## 3.4.2 PWM modes

★ ★

### Subjects

- PWM timer

### Task

Use Timer 4 in Phase Correct PWM mode to generate a PWM signal on PH3. The signal should have a high time of  $75\mu s$  and a period of  $100\mu s$ . Verify the timing of the signal with the oscilloscope.

As an experiment, set the timer prescaler to its highest value (keep all other timer settings – the period will get longer). Again, verify the timing of the new signal with the oscilloscope. If there are any deviations from the expected signal timing, correct them and document this in the protocol.

### Remarks

- Let the timer generate the signal automatically.
- The oscilloscope is accurate well below  $1\mu s$ .
- When you initialize the timer, it is best to first select the mode, then set the registers (like OCR or whatever you need), and last start the timer.

### Questions

1. Which timer modes can you use for this exercise? Which one did you choose?
2. Explain the differences between Phase Correct and Phase and Frequency Correct PWM.
3. How can you generate a Phase Correct PWM signal that is constant (either constantly low or constantly high)?

### Pitfalls

- Do not forget to set PH3 to output.

- Note that the timer counts from 0 to (including!)  $TOP$  (or  $OCRxA$ ) before raising an interrupt, so the duration is  $TOP + 1$  ticks. This is easily overlooked and does not make much difference for a small prescaler, but makes a huge difference for a large prescaler.

### 3.4.3 Generating periodic signals

★ ★

#### Subjects

- Busy-Wait delays
- Timer Overflow
- Output compare timer
- PWM timer

#### Task

You want to generate a periodic signal on PH3 which is high for  $50\mu s$  and low for  $25\mu s$ . Consider the following ways of achieving this signal:

- (1) **Busy-Wait:** In your main loop, you set PH3 to high and enter a busy-wait loop that waits for  $50\mu s$ , then you set PH3 to low and enter another loop that waits for  $25\mu s$ .
- (2) **Timer Overflow:** You set PH3 to low and set up Timer 4 in normal mode to generate an overflow interrupt after  $25\mu s$ . In the overflow ISR, you set PH3 to high and set up Timer 4 to generate an overflow interrupt after  $50\mu s$ . In the next call of the ISR, you set PH3 to low and set the timer to  $25\mu s$  and so on. (Use the register values for  $25\mu s/50\mu s$  here, do not adjust for the latency and ISR code.)
- (3) **Output Compare:** You set PH3 to low and set up Timer 4 in CTC mode (OCR4A) to toggle PH3 after an output compare match and to generate an output compare interrupt after  $25\mu s$ . In the output compare ISR, you alternately set the OCR-register to  $50\mu s$  and  $25\mu s$ .
- (4) **PWM:** You configure Timer 4 to generate a Fast PWM signal using ICR4 as TOP register.

Implement each of these methods (write four programs). For interrupt-driven methods, enter an appropriate sleep mode in main.

#### Remarks

- Do not spend too much time to get method (1) completely accurate. But do try to come close. The point here is to show how hard it is to wait for a specific amount of time in a C busy-wait loop. You may not use the delay loops from the `avr-libc`.
- When you initialize the timer, it is best to first select the mode, then set the registers (like OCR or whatever you need), and last start the timer.

### Questions

1. How long is the signal really high and low in each of these methods and why? How do you compute the actual high- and low-times? Verify your calculations with the oscilloscope.
2. Compare the four methods with respect to their accuracy, their use of hardware resources, and their processor load.
3. What are the shortest periods supported by methods (3) and (4)? What happens if the periods become shorter than these bounds?
4. If you connected a LED to PH3, what effect would you observe (as opposed to simply setting PH3 to high)?
5. Do you have to observe any order when accessing the 16 bit registers in normal and CTC mode?

### Pitfalls

- Do not forget to set PH3 to output (see ATmega1280 manual [2], p. 100).
- Note that the timer counts from 0 to (including!) TOP (or OCR1A) before raising an interrupt, so the interval is TOP+1 ticks. This is easily overlooked and does not make much difference for a small prescaler, but makes a huge difference for a large prescaler.

### 3.4.4 PWM signals and glitches

★ ★

#### Subjects

- PWM timer

#### Task

Write a program that demonstrates what kinds of glitches can occur in each of the Fast PWM, Phase Correct PWM, and Phase and Frequency Correct PWM modes. Do so, write three programs, one for each PWM mode. In them, generate a periodic signal on PH3 that is initially high for  $50\mu\text{s}$  and low for  $25\mu\text{s}$ . Then alter the compare and/or top values appropriately to produce all glitches (changes in the period, high/low time, or symmetry of the signal that neither correspond to the old PWM signal nor to the new one) possible in the given mode.

#### Remarks

- Since this program should demonstrate the glitches, they should be easily and reliably visible on the oscilloscope.

### Questions

1. What kinds of glitches can occur in each of the modes?
2. What did you have to do to reliably produce these glitches in your programs?



## 3.5 Analog module

### 3.5.1 Analog conversion

★ ★

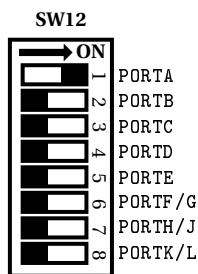
#### Subjects

- A/D converter

#### Task

Use the A/D converter to determine the voltage level of ADC0. Configure the converter for maximum resolution and set the voltage reference to AVCC. Display the highest 8 bits of the conversion result on the LEDs. Use an ISR to react to the conversion complete event and to display the result.

#### Pin assignment



All other switches are set to off.

#### Remarks

- Set **J15** to PF0 and **J18** to VCC.
- The full description of the module is not important for the exercise. Just read the introductory text starting on page 275 and the register description from page 289 on. After you have completed the exercise, read the rest of the module description.

#### Questions

1. Do you have to enable the pull-up for the trimmer?
2. To which value do you have to set the prescaler and why?
3. If you assume that the ADC has the conversion range  $[0,5]V$ , then what is the granularity of the AD converter (i.e., the voltage value of the lsb of the result)? How do you compute this value (give a formula for it)?
4. Assuming no noise, is the lsb (the least significant bit) of the conversion result always correct? Explain why or why not.
5. Assume that for a constant trimmer position the LEDs representing the lowest 4 bits of the conversion result flicker. Can you assume that these four bits are noisy?

**Pitfalls**

- Don't forget to set the reference voltage to AVCC with external capacitor at AREF and to set the prescaler so that you get maximum resolution.
- When you read the ADCL register, the controller does not update the ADC data register until you read ADCH.

**Hints**

- To make things easy, use the single conversion mode and ignore the auto trigger function. You might also want to use the left-justified mode.

**References**

- ATmega1280 manual [2], analog converter: p. 275–295, registers ADMUX, ADCSRA, ADCL, ADCH

**3.5.2 Noise**

★ ★

**Based on:** [3.8.1](#)**Subjects**

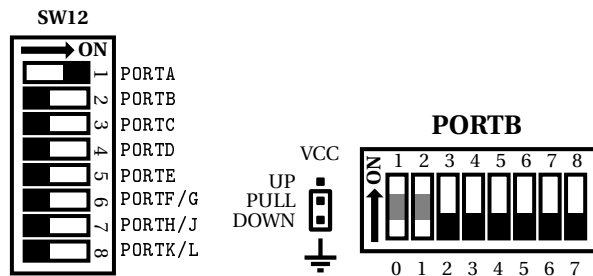
- A/D converter
- Switches

**Task**

Connect the trimmer to pin ADC0 of the controller and use the ADC to determine the voltage level of ADC0. Set the prescaler so that you get maximum resolution. Display the lowest 8 bits of the conversion result on the LEDs. Now we want to see how noise affects the conversion result. Use PB1 : 0 to select different operating modes which control what you do in the main loop:

PB1	PB0	operating mode (action taken in main loop)
0	0	do nothing (except check PB1 : 0)
0	1	enter noise cancellation mode
1	0	set PA1 to digital out and toggle its value in each iteration
1	1	set PA1 to digital in, connect it to PD5, and toggle PD5 in each iteration

Set DDRB1 : 0 to input and PORTB1 : 0 to pull-up, use the external pull-down for switching the modes.

**Pin assignment**

All other switches are set to off.

**Remarks**

- Set **J15** to PF0 and **J18** to VCC.
- Use the single conversion mode.
- Don't enable the LEDs for PORTB, otherwise you'll get problems with the switches.

**Questions**

1. Why do we now display the lowest 8 bit of the conversion result, not the highest 8 bit as in Exercise 3.5.1?
2. Monitor the analog signal on the oscilloscope. How do the modes affect the signal?
3. How do the operating modes affect the conversion result? How many bits are affected in each of the modes?
4. Does the analog value itself influence the accuracy (i.e., can simply changing the trimmer to another value produce a different accuracy)?

**Pitfalls**

- When you read the ADCL register, the controller does not update the ADC data register until you read ADCH.

**3.5.3 Prescaler and accuracy**

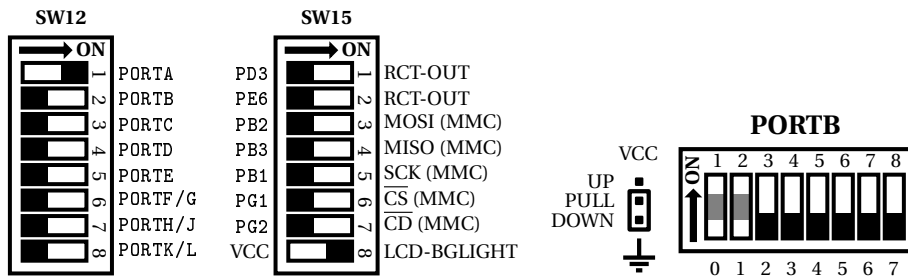
★ ★

**Subjects**

- A/D converter
- LCD

**Task**

Use the A/D converter to read the value from the trimmer. Read every value twice: First trigger a conversion after you have configured the converter to the highest prescaler, then set the prescaler to the setting specified by the DIP switches and trigger a second conversion. Display the result of the first conversion on the first line of the text LCD and the result of the second conversion on the second line of the LCD.

**Pin assignment**

All other switches are set to off.

**Remarks**

- Set **J15** to PF0.

**Questions**

1. What influence does the prescaler setting have on the result? What behavior do you observe? Do a nice presentation (table or graphical) of your findings.

**Hints**

- For displaying numbers on the text LCD you can use the driver from [3.8.3](#) or [3.8.4](#).

**Pitfalls**

- When you read the low byte of the conversion result, it remains frozen until you read the high byte.

**3.6 Communication****3.6.1 UART receiver**

★

Based on: [3.8.3](#)

**Subjects**

- UART
- LCD

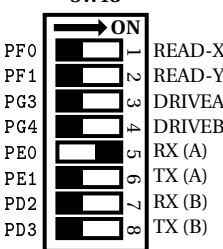
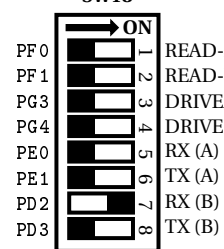
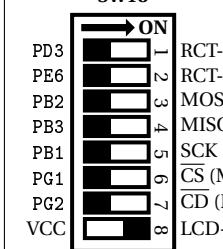
**Task**

Program the receive unit of the UART module to receive characters from the PC and display them on the LCD display. Use asynchronous mode, 8 bit, no parity and 1 stop bit (also called 8N1). You can choose to use UART port A or B.

For sending data between the PC and the microcontroller you can use a program like minicom, gtkterm or cutecom. Alternatively you can also use setserial to set

up the connection settings and directly operate using `echo`, `cat`, `xxd`, ... on the serial device. In the case that your computer has an internal serial port, the device is `/dev/ttyS0`. In case of a USB-to-serial converter it's `/dev/ttyUSB0`.

### Pin assignment

RS232-A	RS232-B	Both
<div><p>SW13</p></div>	<div><p>SW13</p></div>	<div><p>SW15</p></div>

### Remarks

- The LCD character set has a different assignment of special characters like ä, ö, ü, ß compared to the ASCII standard, that means when you send such a special character maybe the display will show something different.
- Use interrupts for the USART, not polling.
- To make your code versatile, it might be wise to implement functions for sending characters and whole strings. To test them, you can for example send a welcome message at the start of the program.
- Read the whole section on the USART in the manual before you begin to program. Use a state diagram to visualize how the asynchronous mode works, what registers you should set at which stages, and which bits are set/which interrupts are raised by the controller.

### Questions

1. What is the difference between a UART and a USART? Come up with some advantages for each of synchronous and asynchronous mode.
2. Which pin is used for the clock signal in synchronous mode? Why do you not need it in asynchronous mode? Who generates the clock signal in synchronous mode? Where does the clock come from in asynchronous mode?
3. Table 22-9 on pages 227/228 of the ATmega1280 manual [2] tell you how to set the UBRR to use a given baud rate.  
How is the UBRR used to generate the baud rate?  
What does the error column in the tables signify and how is it computed?  
Why does a clock of 11.0592MHz not produce any errors for baud rates between 2400 and 230.4kbps, but a clock of 16MHz does?  
Why is 1Mbps not available for the 11.0592MHz clock?
4. Experiment with the baud rate: Which baud rates can you use in your program to communicate (error-free) with the PC? Choose a suitable baud rate for your program and justify your choice.

Pitfalls

- Do not forget to set the communication settings in the serial terminal you have selected.
- Use blocking ISRs for RXC interrupts. Since they are not set back by the call to the ISR, using ISR\_NOBLOCK could generate a stack overflow!

3.6.2 UART sender

★

Based on: 3.9.1, 3.6.1

Subjects

- UART
- Keypad

Task

Extend the UART receiver program (3.6.1) by the send unit of the UART module to send characters to the PC you enter on the external keypad.

Pin assignment

RS232-A	RS232-B	Both
<div>SW13</div> <div><div>PF0</div><div>PF1</div><div>PG3</div><div>PG4</div><div>PE0</div><div>PE1</div><div>PD2</div><div>PD3</div></div> <div><div>→ ON</div><div>1</div><div>2</div><div>3</div><div>4</div><div>5</div><div>6</div><div>7</div><div>8</div></div> <div><div>READ-X</div><div>READ-Y</div><div>DRIVEA</div><div>DRIVEB</div><div>RX (A)</div><div>TX (A)</div><div>RX (B)</div><div>TX (B)</div></div>	<div>SW13</div> <div><div>PF0</div><div>PF1</div><div>PG3</div><div>PG4</div><div>PE0</div><div>PE1</div><div>PD2</div><div>PD3</div></div> <div><div>→ ON</div><div>1</div><div>2</div><div>3</div><div>4</div><div>5</div><div>6</div><div>7</div><div>8</div></div> <div><div>READ-X</div><div>READ-Y</div><div>DRIVEA</div><div>DRIVEB</div><div>RX (A)</div><div>TX (A)</div><div>RX (B)</div><div>TX (B)</div></div>	<div>SW15</div> <div><div>PD3</div><div>PE6</div><div>PB2</div><div>PB3</div><div>PB1</div><div>PG1</div><div>PG2</div><div>VCC</div></div> <div><div>→ ON</div><div>1</div><div>2</div><div>3</div><div>4</div><div>5</div><div>6</div><div>7</div><div>8</div></div> <div><div>RCT-OUT</div><div>RCT-OUT</div><div>MOSI (MMC)</div><div>MISO (MMC)</div><div>SCK (MMC)</div><div>CS (MMC)</div><div>CD (MMC)</div><div>LCD-BGLIGHT</div></div>

Remarks

- If you want to get a new line in the serial terminal send "\r\n"

Pitfalls

- Use blocking ISRs for UDRE interrupts. Since they are not set back by the call to the ISR, using ISR\_NOBLOCK could generate a stack overflow!

3.6.3 SPI

★★

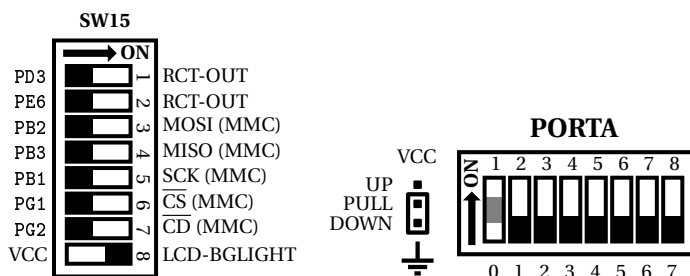
Based on: 3.8.1, 3.8.3

### Task

Write a program that increments a counter whenever the button on PORTA1 is pressed and sends the new counter value over the SPI interface. Simultaneously, your program should read from the SPI interface and display the value it receives as a decimal value on the LC display. With PORTA0, you can select whether your target should be the master (PORTA0 is on) or not (PORTA0 is off). Display the masters value in the first line and the slaves value in the second line.

To test your program, connect two targets and download your program on both of them. On each target, you should see the number of times a button was pressed on the other target.

### Pin assignment



Use jumper wires to connect PPB3:0 of the two targets and set J12 to GND ( $\perp$ ). All other switches are set to off.

### Questions

1. How efficient is the communication (ratio of data bits to all transmitted bits)?
2. Can the slave initiate a communication?
3. Can you set both communication partners to master?
4. What do you have to do to detect transmission errors (bit flips)?
5. Realize the same exercise using the USART in SPI mode.

### 3.6.4 TWI (I<sup>2</sup>C)

★ ★

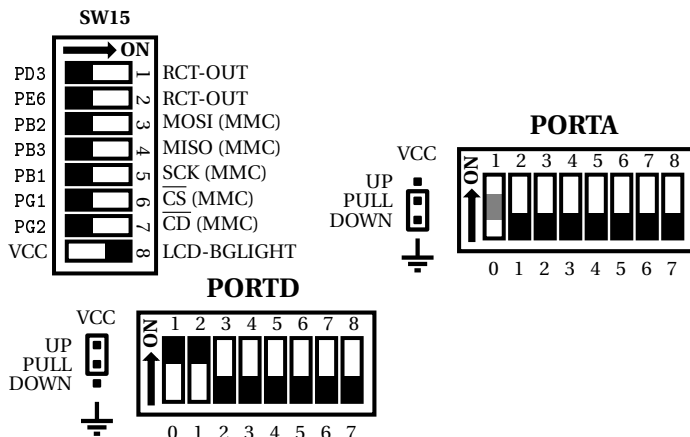
Based on: [3.8.1](#), [3.8.3](#)

### Task

Write a program that increments a counter whenever the button on PORTA1 is pressed and that sends the new counter value over the TWI interface. If your program receives a value over the TWI interface, it should display it as a decimal value on the LC display. With PORTA0, you can select whether your target should be the master (PORTA0 is on) or not (PORTA0 is off). Display the masters value in the first line and the slaves value in the second line.

To test your program, connect two targets and download your program on both of them. On each target, you should see the number of times a button was pressed on the other target.

## Pin assignment



Use jumper wires to connect PPD1 : 0 of the two targets accordingly and set **J12** to GND ( $\perp$ ). All other switches are set to off.

## Remarks

- Select appropriate addresses.

## Questions

1. Why do you need the pull-up resistors on the TWI lines?
2. What is your bit rate?
3. How efficient is the communication (ratio of data bits to all transmitted bits)?
4. Can you set both communication partners to master?
5. What do you have to do to detect transmission errors (bit flips)?

## 3.7 Architecture

### 3.7.1 Calling assembler functions from C

\*\*\*

## Subjects

- Calling assembler functions from C

## Task

Write a C program which maintains a 64bit counter. Use a timer (e.g., timer 1) interrupt with a period of 250ms, increment the counter and write the 64bit to PORTD: A whenever the timer interrupt occurs. Do increment the counter use an external assembler routine. The function signature in C should be:

```
1 extern uint64_t incrementU64(uint64_t counter);
```

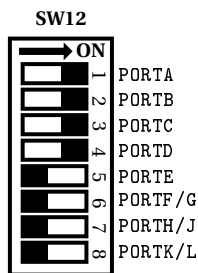
And the complete function in C should look like:

```
1 uint64_t incrementU64(uint64_t counter)
2 {
3     return counter + 1;
4 }
```



Now it's your turn to realize this function in assembler.

### Pin assignment



All other switches are set to off.

### Remarks

- Don't use inline assembler for this task. Use a separate assembler file and link the files together.

### Hints

- The calling conventions in C can be found in [17].
- Alternatively you can also use `avr-gcc -S FILE.c` to transform some C file (FILE.c) to an according assembler file (FILE.s). But be aware of this method, generated assembler files aren't always human-readable.

### Questions

1. When you finished the previous exercise implement

```
1 extern void incrementU64P(uint64_t *counter);
```

with according C function

```
1 void incrementU64P(uint64_t *counter)
2 {
3     ++(*counter);
4 }
```

in assembler.

2. Which of the two types should be preferred? Why have you chosen that one?
3. How would the `const` modifier be realized when calling functions in assembler?

### References

- avr-libc calling conventions, [17]

### 3.7.2 Watchdog

★ ★

Based on: 3.8.1

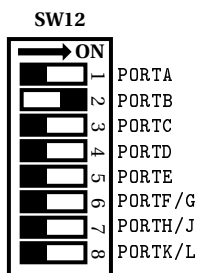
## Subjects

- Switches
- WDT

## Task

Show that the watchdog timer works: Arm the watchdog and use a switch on PA0 to indicate whether the timer should be periodically reset by your program (PA0 is ON) or not (PA0 is OFF). Use a LED on PB0 to show that the watchdog timer works, and integrate the Watchdog Reset Flag into your solution. Use an appropriate timeout period.

## Pin assignment



All other switches are set to off.

## Remarks

- Forget about power consumption issues in this exercise and just poll PA0 in an infinite loop.

## Questions

1. Name two situations in which a watchdog is useful.
2. Discuss how the watchdog helped or did not help in the 1997 Mars Pathfinder mission.
3. How can you disable the watchdog? Why is the procedure so complicated?
4. How can you determine at the start of your program whether the user has pressed the RESET button?

## Pitfalls

- Do you need the internal pull-up for PA0?
- Be aware that the Watchdog Reset Flag is only cleared by a power-on reset, not by an external reset via the reset button.

## Hints

- To be able to verify that the watchdog works, you have to do something at the start of your program, like turning on a LED for some time.

## References

- ATmega1280 manual [2], watchdog timer: p. 63–67

### 3.7.3 Pipelining

★ ★

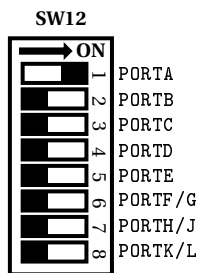
#### Subjects

- Instruction execution

#### Task

On page 17 of the ATmega1280 manual [2], it is stated that the AVR CPU pipelines instruction fetches and executions to obtain up to 1MIPS per MHz. Demonstrate or refute this claim using a LED on PORTA and the oscilloscope.

#### Pin assignment



All other switches are set to off.

#### Questions

1. What is the idea behind your program?
2. What did you find out?
3. Are the cycle numbers given in the Instruction Set Summary of the ATmega1280 manual [2] with pipelining or without?

### 3.7.4 Memory considerations

★ ★

#### Subjects

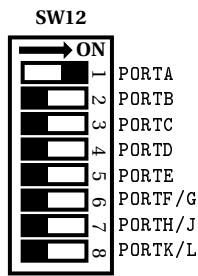
- Flash / Progmam
- SRAM
- EEPROM

#### Task

Display the numbers  $0 \dots F$  as gray code one by one for an appropriate time on PORTA (start again with 0 after you are done). Use a table to store the values for the numbers.

Now create three versions of your program which differ in the location of the table in memory: Your first program should have the table in SRAM, your second in the Flash, and your third in the EEPROM.

## Pin assignment



All other switches are set to off.

## Remarks

- To make things easy, use a busy-wait loop to implement the delay.
- Remember that the SRAM cannot be initialized automatically, you have to initialize the table within your program.
- Note that the controller gets halted by access to the EEPROM.

## Hints

- The include file `avr/pgmspace.h` can be interesting for reading flash memory. Especially take a look at `PROGMEM`, `PSTR` and `pgm_read_byte`.

## Questions

1. Compare the three programs with respect to ease of setting up the table (definition and initialization) and of accessing it in the program.
2. How fast (in system clock cycles) can you read a table entry if it is in SRAM? How fast if it is in the Flash? How fast if it is in the EEPROM? Include everything that belongs to the read operation, like writing to the address registers in case of the EEPROM.
3. How fast can you write a table entry in SRAM? How fast in EEPROM? Why do we not ask about Flash writes?
4. For how long did you display each number? Could you display them for exactly one second? If yes, then demonstrate how you would do it, if not, explain why not.

## References

- ATmega1280 manual [2], Memories: p. 21–27

### 3.7.5 Dynamic memory analysis

★ ★

## Subjects

- SRAM

**Task**

As you may know, static memory analysis can be done quite easy using `avr-size`, dynamic memory analysis isn't that easy cause you have to analyze a running program. For dynamic memory analysis we exploit that the AVR microcontrollers don't have a memory management unit and we can read and write the whole memory. To get the dynamic memory usage we have to initialize the whole RAM with a specific pattern like 0x55, 0x21 or 0x41 which isn't used by the program during execution before any routine is executed. Then run all your routines of your program (including the execution of all possible interrupts) and check the memory afterwards if it still contains the specified pattern. Count every register which has a different value than the specified pattern and display it on the LCD or send it via UART. In reality you can't just overwrite the whole RAM, you have to start at the end of the so called BSS section and stop at the stack pointer. Otherwise initialized variables and program counters on the stack will be destroyed.

$$\text{dynamic memory} = \frac{|\text{different registers}|}{|\text{total memory}|}$$

**Hints**

- Use following to get the end of the BSS section in C:

```
1 uint16_t bss;
2 asm volatile("ldi %A0, lo8(__bss_end)\n ldi %B0, hi8(
   __bss_end)" : "=r" (bss));
3 (uint8_t *) bss;
```

- Use following to get the stack pointer (SP) in C:

```
1 (uint8_t *) SP;
```

**3.7.6 CPU usage**

★★★

**Subjects**

- Instruction execution

**Task**

Measure the CPU usage of your program using the method of counting unused cycles of the microcontroller. You will need a 32bit counter and a timer. Configure a 16bit timer with a cycle time of something between 1 and 4s. In the main loop of your program increment the counter by one and after every timer interrupt send the value to the computer using the UART or display it using the LCD, then reset the counter. To calculate the CPU usage correctly you have to disassemble the source code and see how many cycles it takes to increment the counter by one and how long the timer interrupt needs for correcting that.

If you have a good value of free cycles you can calculate the CPU usage using following formula:  $\%_{CPU} = \frac{\text{free cycles}}{\text{cycles per timestep}}$  where timestep denotes the time between two timer interrupts.

### Remarks

- For this task you have to disable any sleep modes.
- Don't forget to reset the main-loop counter.

## 3.8 On-board hardware

### 3.8.1 Switches

★ ★

#### Subjects

- Digital I/O
- Switches

#### Task

Since there are only push buttons and no explicit switches on the board we use the pull-down switches for that purpose. That works the following way: set the wanted port to input and enable the internal pull-up. Then set the external pull-switches for that port to pull-down. Now you can use them for switching anything on (low) or off (high).

#### Pitfalls

- Don't enable the LEDs for that port otherwise it won't work.

### 3.8.2 Button debouncing

★ ★

#### Subjects

- Digital I/O
- Buttons
- External interrupt

#### Task

Write an ISR using any external interrupt between 0 and 7 that increments a counter (modulo 10) whenever the button is pressed and that displays this counter value on PORTA. Take counter-measures to prevent bouncing.

#### Remarks

- Bouncing is a side-effect and unpredictable. It may happen that you have a board where a button doesn't bounce (yet). In that case, try another button. Chances are good that you will quickly find a button that does bounce. For this exercise, use the button that bounces most.

## Questions

1. How can you debounce a button using hardware?
2. Examine the bouncing signal with the oscilloscope and try to capture bouncing on the scope. How long does it generally take until the signal is stable? What is the maximum bounce time you observed? How often does the signal bounce before it stabilizes? Does it only bounce when pressed, or when released, or in both situations?
3. Describe the debouncing strategy you have implemented. How does it work? Which assumptions, if any, does it make?
4. List at least two other (significantly different) methods to debounce the button and explain how they work. Compare them to your method in terms of ease of use, coverage, and processor load.

## Pitfalls

- If your solution disables the external interrupt for some time, do not forget to clear the interrupt flag before enabling the interrupt again.

### 3.8.3 Using the precompiled LCD module

★

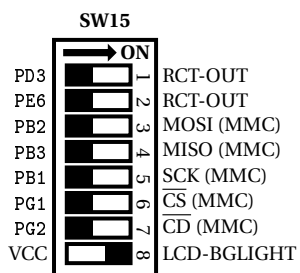
## Subjects

- LCD

## Task

Download [18], look at the `main.c` and `drivers/lcd.h` to get known to the LCD driver.

## Pin assignment



All other switches are set to off.

## Provided functions

- Initialize the port and the LCD. Call this before any other LCD function is called.

```
1 void initLcd(void)
```
- Synchronizes the screen. Sends exactly one command or one character per call. Be aware that you wait at least 2ms between calls.

```
1 void syncScreen(void)
```

- Clears the framebuffer, synchronize the screen afterwards.

```
1 void clearScreen(void)
```

- Writes a string to the specified position in the framebuffer.

```
1 void dispString(char *str, uint8_t x, const uint8_t y)
```

- Writes an unsigned 8 bit number (uint8\_t) to the specified position in the framebuffer.

```
1 void dispUInt8(uint8_t num, uint8_t x, uint8_t y)
```

- Writes one character to the specified position in the framebuffer.

```
1 void dispChar(char c, uint8_t x, uint8_t y)
```

- There's also a file-descriptor provided with which you can write to the framebuffer using things like `fprintf`. You can step to the first line and character by writing `'\r'` and jump into the next line using `'\n'`. When `'\n'` is read, the line is filled up with spaces (so you can terminate the last line with `'\n'` to get rid of unwritten characters).

```
1 FILE *lcdout;  
2 fprintf(lcdout, "Some formatted %s", "text");
```

### Remarks

- In this function reference `const` modifiers were omitted.

### 3.8.4 LCD

★★

#### Subjects

- LCD

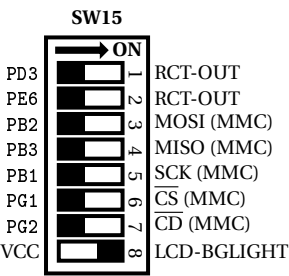
#### Task

Write a simple LCD driver which uses a framebuffer. The driver should work the following way: You have an internal framebuffer which stores the whole screen content in a char-array and when calling a synchronize method the framebuffer gets sent to the LCD. Whenever you want to write text to the screen, it gets written into the framebuffer. After the next synchronization it's displayed on the screen. The screen should be synchronized with about 4 – 2Hz.

The driver should be able to write characters, numbers (decimal and hexadecimal) and text to the framebuffer at a specific  $x/y$  position ( $x \in \{0, \dots, 15\}$ ,  $y \in \{0, 1\}$ ).



Pin assignment



All other switches are set to off.

Remarks

- The manual for the text LCD controller can be found by searching "KS0066" in the internet.
- You have to use the display in 4-bit mode.
- You can use the header file from the precompiled LCD module if it helps.
- There's some kind of mistake in the manual, instead of sending

Function set									
RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	1	0	X	X	X	X
0	0	0	0	1	0	X	X	X	X
0	N	F	X	X	X	X	X	X	X

it's better to send

Function set									
RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	1	0	X	X	X	X
0	N	F	X	X	X	X	X	X	X
0	0	0	0	1	0	X	X	X	X
0	N	F	X	X	X	X	X	X	X

when initializing.

3.8.5 Using the precompiled Graphic GLCD module

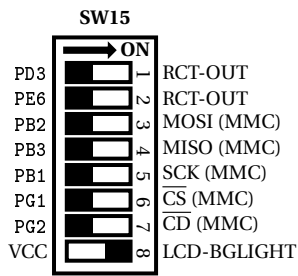
★ ★

Subjects

- GLCD

Task

Download [19], look at the `main.c` and get known to the GLCD driver.

**Pin assignment**

All other switches are set to off.

**Provided functions**

- Initialize the port and the GLCD. Call this before any other GLCD function is called.

```
1 void initGLCD(void);
```

- This function fills the screen with a specified pattern. To get a white screen use 0xFF and for a black screen use 0x00.

```
1 void fillGLCDScreen(uint8_t fill);
```

- Sets a pixel on given  $x/y$  position. Given pixel is white afterwards.

```
1 void setPixelGLCD(uint8_t x, uint8_t y);
```

- Clears a pixel on given  $x/y$  position. Given pixel is black afterwards.

```
1 void clearPixelGLCD(uint8_t x, uint8_t y);
```

- Inverts a pixel on given  $x/y$  position.

```
1 void invertPixelGLCD(uint8_t x, uint8_t y);
```

- Draws a line from  $p1$  to  $p2$  using given drawing function. The Bresenham algorithm is used for line interpolation.

```
1 void drawLine(xy_point p1, xy_point p2, void (*drawPx)(  
    uint8_t, uint8_t));
```

To draw a white line from (10/10) to (30/30) execute following:

```
1 xy_point p1, p2;  
2 p1.x = p1.y = 10;  
3 p2.x = p2.y = 30;  
4 drawLine(p1, p2, &setPixelGLCD);
```

- Draws a rectangle from  $p1$  to  $p2$  using given drawing function. Only the outline is drawn, for a filled rectangle see fillRect.

```
1 void drawRect(xy_point p1, xy_point p2, void (*drawPx)(  
    uint8_t, uint8_t));
```

- Draws a filled rectangle from  $p1$  to  $p2$  using given drawing function.

```
1 void fillRect(xy_point p1, xy_point p2, void (*drawPx)(  
    uint8_t, uint8_t));
```

- Draws a circle with center  $c$  and radius  $radius$  using given drawing function. The circle Bresenham algorithm is used for interpolating points on the circle.

```
1 void drawCircle(xy_point c, uint8_t radius, void (*drawPx)(
    uint8_t, uint8_t));
```

- Draws an ellipse with center `c` and the two radii `radiusX` and `radiusY` using given drawing function. An ellipse Bresenham implementation is used for interpolating points on the ellipse.

```
1 void drawEllipse(xy_point c, uint8_t radiusX, uint8_t
    radiusY, void (*drawPx)(uint8_t, uint8_t));
```

- Draws a vertical line on given `x` position on the screen. As always you can specify a specific drawing function.

```
1 void drawVertical(uint8_t x, void (*drawPx)(uint8_t, uint8_t
    ));
```

- Draws a horizontal line on given `y` position on the screen using given drawing function.

```
1 void drawHorizontal(uint8_t y, void (*drawPx)(uint8_t,
    uint8_t));
```

- Draws a given character `c` on position `p` using given drawing function and font. The anchor for drawing characters is on the bottom left of the drawn character.

```
1 void drawChar(char c, xy_point p, font* f, void (*drawPx)(
    uint8_t, uint8_t));
```

- Draws a given text on position `p` on the screen. The font and drawing function has to be specified. The anchor is again bottom left of the text.

```
1 void drawText(const char *text, xy_point p, font* f, void (*
    drawPx)(uint8_t, uint8_t));
```

- You can specify your own fonts (maybe symbols) using the datatype given in `font.h`. The character width is limited by 255 and the height is limited by 8

```
1
2 static const uint8_t MyFontCharacters[] PROGMEM =
3 {
4     0x20, 0x54, 0x54, 0x54, 0x78,    // a
5     /* ... Characters between a and z ... */
6     0x44, 0x64, 0x54, 0x4C, 0x44,    // z
7 };
8
9 const font MyFont = {'a', 'z', 5, 7, 6, 8, MyFontCharacters
    };
```

## Remarks

- In this function reference `const` modifiers were omitted.
- The precompiled GLCD driver uses busy-waiting.
- You could even use your own drawing function (when implementing your own GLCD driver).

### 3.8.6 Graphic LCD

★ ★

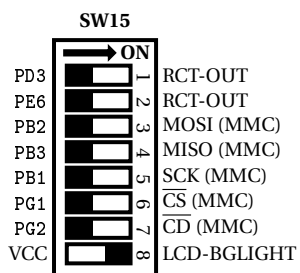
#### Subjects

- GLCD

#### Task

Write a small library which implements the most important drawing functions like setting a point, clearing a point and inverting and test it in a small program. To test your functions you can use the drawing methods (`drawLine`, `drawRect`, ...) from the pre-compiled LCD module with your drawing functions.

#### Pin assignment



All other switches are set to off.

#### Remarks

- The manual for the text LCD controller can be found by searching "KS0108" in the internet.

### 3.8.7 Touch panel

★ ★

Based on: [3.5.1](#)

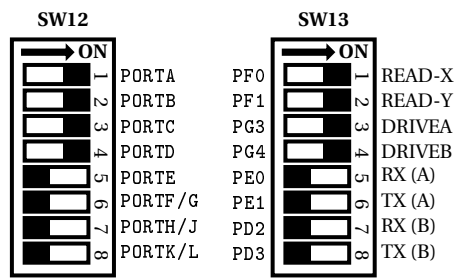
#### Subjects

- A/D converter
- Touch panel

#### Task

Write a small driver for the touch panel which uses the A/D converter driver to read the values from PF0 and PF1 and display the LEFT value on PORTA0 : 8 & PORTB0 : 1 and the BOTTOM value on PORTC0 : 8 & PORTD0 : 1. For converting the LEFT value you have to set DRIVEA to high and DRIVEB to low and for the BOTTOM value set DRIVEA to low and DRIVEB to high. The result can be improved by calculating the mean of 4 conversions. Furthermore write a method to check if the touchscreen is pressed. Implement this using some thresholds for  $x$  and  $y$ .

## Pin assignment



All other switches are set to off.

## 3.9 External Hardware

### 3.9.1 Keypad

★ ★

Based on: [3.8.3](#)

#### Subjects

- Digital I/O
- Keypad
- LCD

#### Task

Attach the keypad module to PORTA. Write a program which handles the input of the keypad.

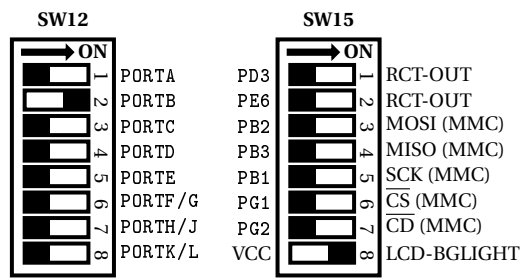
Set the lower 4 pins of PORTA to output and the upper 4 pins to input with pull-up. To read a button from the keypad you have to switch through the rows and read the columns. That works the following:

- Set column 1 (PA0) to low.  $PA3:0 := 1110$
- Read row (PA7 : 4), if a pin is low then according button in column 1 is pressed.
- Set column 2 (PA1) to low.  $PA3:0 := 1101$
- Read row (PA7 : 4), if a pin is low then according button in column 2 is pressed.
- ...

If every row is high then no button is pressed.

Display the pressed button in the main loop on the LCD.

To improve the keypad readout use a timer with a period of about 50ms. At every call of the timer ISR read the row and activate the next column. Save the pressed key to a (volatile) global variable to read it in any other ISR or main loop.

**Pin assignment**

All other switches are set to off.

**Remarks**

- You can swap rows and columns since there are no diodes on the board.

**References**

- 4x4 keypad manual [\[20\]](#)

**3.9.2 Humidity & Temperature (SHT1X)**

★ ★ ★

Based on: [3.6.4](#)

**Subjects**

- LCD
- SHT1X

**Task**

Attach the SHT1X extension board to PORTD and set the switches to AVR. Write a program which reads the temperature and humidity from the SHT1X over TWI and display the values on the LCD. Convert values every 5s.

**References**

- SHT1X extension board manual [\[21\]](#)
- SHT1X datasheet [\[22\]](#)

**3.9.3 Light to frequency**

★

Based on: [3.4.1](#)

**Subjects**

- LCD
- Input capture
- Light to frequency

**Task**

Use the input capture function of Timer 4 to measure light. For this task attach the "Light2Frequency" (L2F) module to PORTL, set Timer 4 to normal operation (free-running). In a room with normal lighting (no direct sun light) it's best to use the L2F module with a sensitivity of 1 ( $S1=0, S0=1$ ) and a frequency scaling of 10 ( $S3=1, S2=0$ ) and the timer with a prescaler of 64 (assuming 16MHz clock). To get correct values even in dark rooms count the overflows in the timer overflow interrupt and use them in period calculation. ( $CNT_{PERIOD} = ICR_{current} + OVW * 0xFFFF - ICR_{old}$  where CNT contains the timer steps between a period). Use the LCD module with an update frequency of about 500Hz to display the last measured period.

**Questions**

1. Does it make a difference of using rising or falling edge in this task?
2. Can you also use input capture in combination with ...?
  - (a) CTC
  - (b) Fast PWM
  - (c) Phase correct PWM
  - (d) Phase & frequency correct PWM

Argue why you can or can't use it!

**References**

- Light 2 Frequency manual [23]

**3.9.4 MMC card**

★ ★

Based on: [3.6.3](#), [3.8.3](#)

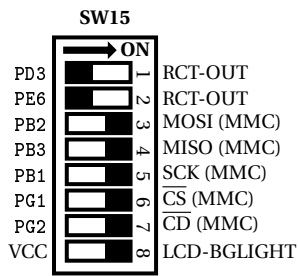
**Subjects**

- SPI
- LCD

**Task**

Write a program which displays the directory listing of a FAT16 formatted SD card on the LC display. For simplifying this task only display the files contained in the SD card's root directory. Since the LC display has only two lines to display text on, use two buttons for switching to the next file (PORTA0) or to the previous one (PORTA1). Use a third button (PORTA2) to open the selected file and display the filename in the first line and the first 16 characters of the file's content on the second line of the LC display. To get back to file selection press (PORTA3).

### Pin assignment



All other switches are set to off.

### Remarks

- For this task use the onboard MMC / SD card reader.

### 3.9.5 RFID

★ ★ ★

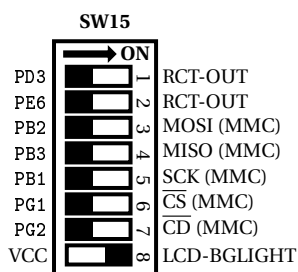
### Subjects

- RFID
- LCD

### Task

Write a program which displays the ID of the RFID card which is held onto the RFID reader. To read data from the RFID reader attach it to the port you prefer and set OUT and RDY/CLK to input and SHD and MOD to output and low. The RFID cards in the lab should be manchester encoded with 64 cycles / bit. You have to decode the manchester line code by yourself. How the received bit sequence looks like is stated in [?] on page 5. One interrupt driven approach for doing this is to register an external interrupt (INT0) for any logical change and measure the time between the edges. To use INT0 for sensing the edges you'll have to attach the RFID reader to PORTD.

### Pin assignment



All other switches are set to off.

### Remarks

- It's a good idea to check the checksum because wrong signals could be received quite often.



For a parity check you can use the method from `util/parity.h` or generate it on the fly during RFID card reading.

## References

- RFID card manual (EM4100) [?] ]
- RFID extension module manual [?] ]
- EM4095 datasheet [?] ]

## 3.10 Applications

### 3.10.1 Random number generator

★ ★

Based on: [3.8.3](#)

## Subjects

- LCD
- External interrupt
- Random

## Task

Implement a random number generator which generates random numbers on demand. Install an external interrupt on INT7 and whenever the button is pressed generate a new number and display it on the LCD.

**Physical random numbers** You can read values from the A/D converter and use the lower bits (noise) of the conversion for generating a random number.

**Pseudo-random numbers** You can use following code segment for generating pseudo random numbers. The sequence repeats after 127 values. There are 18 different primitive polynomials of degree 7 which could be used (with a period of 127). The following snippet realizes the primitive polynomial  $p(x) = x^7 + x + 1$ .

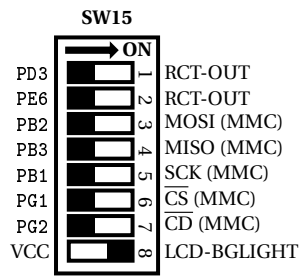
```
1 uint8_t x = 0x42;    /* some initial value != 0 */
2 x = (x<<1) | ((x & (1<<7))>>7) ^ ((x & (1<<1))>>1) ^ (x & (0x01))
    ;
```

To get a period of 255 you'd have to use a primitive polynomial with a degree of 8.

**Better random numbers** There also exists a paper about generating real random numbers on AVR microcontrollers, search for [\[24\]](#).

## Remarks

- It's better to implement a some real random number generation and not only the pseudo-random number (PRN) generator because for later usage (e.g., [3.11.1](#)) the pseudo-random numbers will repeat after some period and always start with the same value which won't fulfill the requirements for a dice.

**Pin assignment**

All other switches are set to off.

**References**

- See [25] and [26] for more information.

**3.10.2 GPS 1PPS**

★ ★

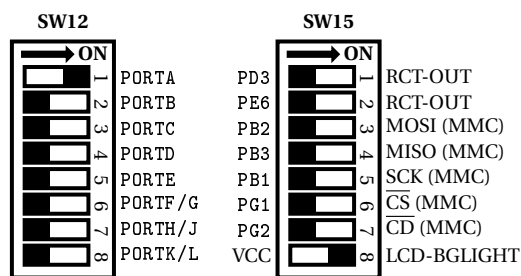
**Subjects**

- GPS

**Task**

Write a program which blinks a LED (PORTA0) with exactly 1 pulse per second. Capture the time data from the GPS. If you have no GPS use the GPS simulator from [27]. If you have access to a real GPS module you could also use the 1PPS pin for this task.

GPS data consists of several NMEA 0183 data lines, for this task the most important lines are GPGLA and GPRMC because they contain the UTC timestamp.

**Pin assignment**

All other switches are set to off.

**3.10.3 Alarm clock**

★ ★ ★

Based on: **3.10.2** or ??

**Subjects**

- GPS or RTC
- Timer
- LCD

- Keypad
- Switches

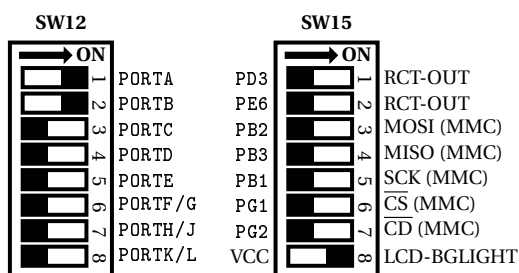
### Task

Program a simple alarm clock which displays the current time and has one programmable alarm. To get the time use GPS data or the RTC. Use a switch for configuring the alarm clock. If PORTD0 is ON, the alarm is configurable by 3 buttons, PORTD2 increases the selected number by 1, PORTD3 decreases the number by 1, PORTD4 steps to the other number (hours → minutes, minutes → hours). If PORTD1 is ON the alarm is active. The alarm clock should always display the current time in the upper line on the LCD and the alarm time on the lower line if it's activated. In the case that the alarm is deactivated display a 'X' on the end of the second line. In configuration mode display a 'C' on the end of the first line. When the clock time meets the alarm time an alarm is executed and the LEDs of PORTA and PORTB should flash for 30s.

**RTC** When using the RTC for getting the time you'll have to set the correct time first.

**GPS** For the GPS you have to consider that time is given in UTC time and not CET/CEST do don't forget to add 1 or 2 hours.

### Pin assignment



All other switches are set to off.

### Remarks

- Be aware to use the right sleep mode and interrupts (clocks should save energy).
- You can also use the GLCD instead of the LCD and display a clock face with two hands on the screen.
- You can also extend the clock by using the MP3 player for playing some alarm sound.

### 3.10.4 Scrolling text

★ ★

Based on: [3.8.4](#)

### Subjects

- LCD

### Task

When displaying text which is longer than the LCD width you normally cut it off. Now you should implement a scrolling text feature to display text which is up to 255 characters long. Start with displaying the text at position 0 and scroll it to the left every with a frequency of 1Hz. If the end of text meets the right end of the display you should wait for 2s and then scroll back to the left.

### Remarks

- The maximum text length is 255 but the char-array has to be 256 characters long (because of terminating '`\0`').

### 3.10.5 Distance sensor

★ ★ ★

Based on: [3.9.3](#), [3.8.3](#), [3.8.2](#), [3.8.1](#)

### Subjects

- Light to frequency
- Input capture
- LCD
- Button debouncing
- Switches

### Task

Our hardware does not include a distance sensor. However, when you hear someone propose to use the photo diode (in our case the photo diode array from the L2F module) for this purpose, arguing that as long as an object casts more shadow on the sensor the nearer it comes, one can use this fact to compute the distance, you are intrigued and decide to implement this idea to check its merits.

Modify the program from [3.4.1](#) to use the L2F module for computing the distance of an object. Display the distance in cm on the LCD. To compute the distance, use a mapping table with about 5–10 points and interpolate between these points. Include a calibration mode in your program to set up the mapping table. For each point in the table, the program should display its distance on the LCD (display a '`C`' on the screen to indicate that this is the calibration mode). Then wait until the user presses PA1 and store the current value of the photo transistor in the table. Iterate through all points in the table in this way. To enter calibration mode, the user should switch on PA0. The calibration mode is left after the last point has been entered.

### Remarks

- Again attach the L2F module to PORTL.
- We are aware that it is not so easy to get a good distance sensor out of the L2F module, but we rely on your ingenuity to make this work (at least under some special conditions, for a demonstration to your friends).

- You can decide for yourself which range would be best for the sensor. It should be at least a couple of centimeters (10cm or more).
- It's necessary to debounce the used buttons, otherwise calibration is impossible when the buttons are bouncing. As an alternative you could use one button for one distance (so use PA1 for distance 1, PA2 for distance 2, and so on).

### Questions

1. How well does the idea work? Do you have any suggestions how to make it work better?

### 3.10.6 Correct LED dimming

★ ★ ★

Based on: [3.5.1](#), [3.8.3](#)

### Subjects

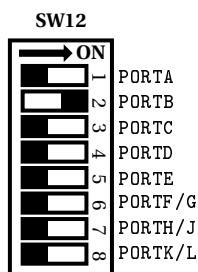
- Correct LED dimming
- A/D converter
- LCD

### Task

Convert the value from the trimmer and use the upper 8bit. To get a stable value take the mean of 8 conversions and discard the value from the first conversion. Display the converted value on the LCD and dim the LED according to the analog value. Use timer 5 with PWM (A, B, C — PORTB7 : 5) to dim the LEDs.

Dimming LEDs with PWM is better than dimming LEDs with analog voltage but even that is not completely linear in humans eye. To get a visual linearity correct the light using Stevens' Power Law:  $\psi(I) = kI^a$ . When using  $a = 0.50$  you can see that the minimum for  $I = 0x00 = 0$  and the maximum is  $I = 0xFF = 255$ . The integer root for  $\lfloor \sqrt{255} \rfloor = 15$  so you can use a  $k$  of about 17 ( $15 \cdot 17 = 255$ ) which results in  $\psi(I) = 17 \cdot \sqrt{I}$ .

### Pin assignment



All other switches are set to off.

### Remarks

- Choose some  $a \in [0.45, 0.50]$  for dimming the LEDs. Probably  $a = 0.5$  will be the easiest value to calculate (you could implement finding the square root using

nested intervals or the Babylonian method).

## References

- See [28] for more information.

### 3.10.7 Sinusoidal signal

★ ★ ★

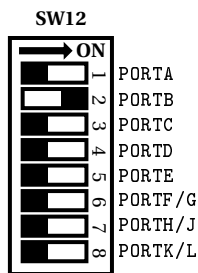
## Subjects

- PWM

## Task

Program a frequency generator which generates a sinusoidal signal. Output the generated signal on the LEDs (using PWM with timer 5 A, B, C — PORTB7 : 5). Find a method to implement the function generator that you can control the amplitude (between 0–5V) and the frequency (between 20–200Hz) using the trimmer.

## Pin assignment



All other switches are set to off.

## Hints

- Using floating points is very slow, so implement everything using integer arithmetic.
- Calculating the sinus signal using the `sin` from `libc` can be very slow so maybe it's better to use a precalculated look-up table with enough values.

### 3.10.8 RFID lock

★ ★ ★

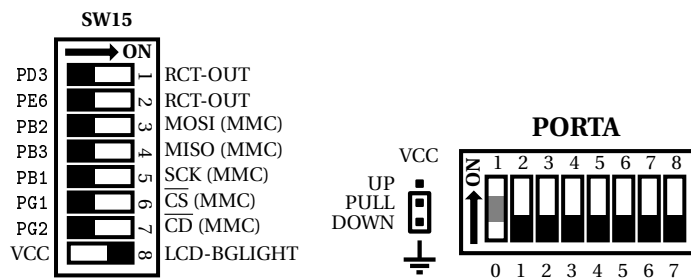
Based on: [3.9.5](#), [3.8.1](#)

## Subjects

- RFID
- LCD
- Switches

**Task**

Modify the RFID reader [3.9.5](#) that it manages a door lock with a maximum of 5 cards. Use PORTA0 as switch, when the switch is ON it's in the programming mode and if it's OFF the normal mode is active. When the lock is in programming mode it waits for a button press on PORTAx  $x \in 1 \dots 5$  and stores the current card on the according position in the database. When in normal mode and a card is held onto the card reader it displays "OPEN" if the card is valid and "LOCKED" if the card is not valid or there's no card on the card reader.

**Pin assignment**

All other switches are set to off.

**3.10.9 Hygrometer**

★ ★

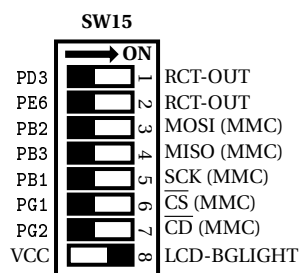
Based on: [3.9.2](#), [3.8.5](#)

**Subjects**

- SHT1X
- GLCD

**Task**

Again attach the SHT1X extension board to PORTD. Modify the temperature & humidity program [3.9.2](#) that it displays the history of the temperature and humidity on the GLCD. Assume senseful lower and upper values to get a nice drawing. Draw a diagram for the conversions on the GLCD.

**Pin assignment**

All other switches are set to off.

**3.10.10 Analog safe lock**

★ ★

Based on: [3.5.1](#), [3.8.3](#)**Subjects**

- A/D converter
- LCD
- Timer

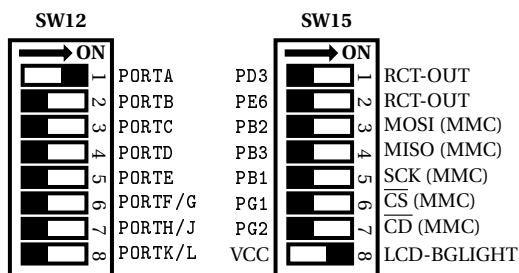
**Task**

Program a safe lock. The safe combination consists of three 2-digit numbers which are defined as constants.

Use the trimmer to enter the combination and unlock the safe. During the whole operation a LED on PA0 shows if the lock is opened or closed.

To implement the safe, start with a locked safe. The display shows "\*\*\* - \*\* - \*\*\*" (safe locked) waiting for action. To start input turn the trimmer to its end positions within 500 ms. Now the display shows "XY - \*\* - \*\*\*" ( $X, Y \in [00 - 99]$  is depending on the trimmer). If the display value does not change during 1s it will be taken over. The display changes to "XY - AB - \*\*\*" ( $X, Y$  are fixed now and  $A, B \in [00 - 99]$  is depending on the trimmer). Now you turn the trimmer and enter the second number by waiting. After the third the display changes to "OPEN" if the code was correct or to "LOCKED" if the unlock failed. After 3s the lock changes back to waiting mode ("\*\* - \*\* - \*\*\*"). Don't forget to lock again.

Define the time constant (1s) global and find good values for a safe and quick unlock procedure.

**Pin assignment**

All other switches are set to off. Furthermore set **J15** to PF0 and **J18** to VCC.

**Questions**

1. Extend the safe by a programming procedure for changing the combination.

**3.10.11 Light adaption**

★ ★

Based on: [3.9.3](#)



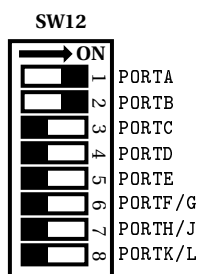
### Subjects

- LEDs
- Light to frequency

### Task

Program a simple light adaption which changes the brightness of LEDs on PORTA7:0 and PORTB7:0 according to the surrounding light. This task has some practical usage in modern smart phones which adjust the screen brightness according to the ambient light that your eyes don't hurt when looking into a bright screen in the darkness. Use the L2F module for getting the ambient light and find a formula to adjust the light correctly.

### Pin assignment



All other switches are set to off.

### Remarks

- Use an adequate sleep mode and do everything in interrupt handlers.
- PWM won't work for the whole port, you can use a CTC interrupt instead.

### 3.10.12 Dew point

★ ★

Based on: [3.8.3](#), [3.9.2](#)

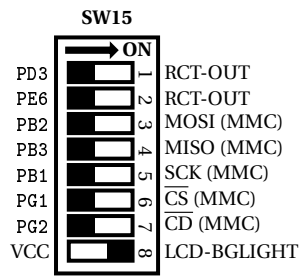
### Subjects

- SHT1X
- LCD

### Task

Attach the SHT1X extension board to PORTD. Modify the temperature & humidity program [3.9.2](#) that it calculates the dew point and displays it on the LCD. You'll find the calculation of the dew point in [\[22\]](#) in section 4.4. Start the conversions every 5s and display the temperature, humidity and the dew point on the LCD.

### Pin assignment



All other switches are set to off.

### References

- SHT1X extension board manual [21]
- SHT1X datasheet [22]

### 3.10.13 Six's thermometer

★ ★

Based on: [3.8.3](#), [3.9.2](#)

### Subjects

- LCD
- SHT1X or DS18S20

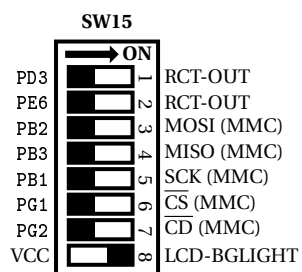
### Task

Program a digital Six's thermometer. For this task you can choose between SHT1X or the DS18S20 for getting the temperature. Always display the minimum, the maximum and the current temperature on the LCD.

**SHT1X** When using the SHT1X convert the temperature every 2.5s.

**DS18S20** When using the DS18S20 convert the temperature as often as possible.

### Pin assignment



All other switches are set to off.

## 3.10.14 Bang-bang controller

★ ★

Based on: 3.8.3, 3.5.1

## Subjects

- LCD
- ADC

## Task

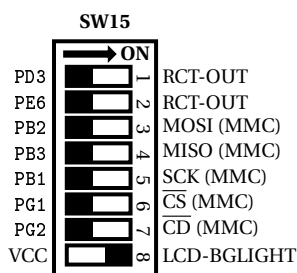
Program a bang-bang controller which reads the current temperature (in our case the 8bit value of the trimmer) and controls the boiler's heating element (in our case an arrow is displayed on the LCD if the user should turn the trimmer right (heating-up) or turn the trimmer left (cooling)). The bang-bang controller needs a lower, an upper value and the current state. In our case the sensor measurement is denoted by  $y_k \in [0, 255]$ , the control vector  $u_k \in \{\text{NOTHING}, \text{HEATING}\}$  and the state  $x_k \in \{\text{NOTHING}, \text{HEATING}\}$ . The lower threshold  $y_{\text{LOW}}$  and the upper threshold  $y_{\text{HIGH}}$ . The next state is given by

$$x_{k+1} = \begin{cases} \text{NOTHING} & y_k > y_{\text{HIGH}} \\ \text{HEATING} & y_k < y_{\text{LOW}} \\ x_k & \text{otherwise} \end{cases}$$

Define the upper and the lower threshold as constant and update the controller value with a specific sampling frequency (e.g., 1kHz). Display the current state and the according symbol on the LC display. HEATING corresponds to  $\rightarrow$  or  $\uparrow$  and NOTHING corresponds to  $\leftarrow$  or  $\downarrow$ .

Use "\xc8" ( $310_8 = 200_{10}$ ) for a left arrow ( $\leftarrow$ ), "\xc7" ( $307_8 = 199_{10}$ ) for a right arrow ( $\rightarrow$ ), "\xc5" ( $305_8 = 197_{10}$ ) for an up arrow ( $\uparrow$ ) and "\xc6" ( $306_8 = 198_{10}$ ) for a down arrow ( $\downarrow$ ).

## Pin assignment



All other switches are set to off. Furthermore set **J15** to PF0 and **J18** to VCC.

## Questions

1. How high has the hysteresis ( $y_{\text{HYS}} = y_{\text{HIGH}} - y_{\text{LOW}}$ ) to be that the noise won't affect the controller? (Find that value out by experimenting with different thresholds)

**3.10.15 (Egg) timer**

★ ★

Based on: [3.5.1](#), [3.8.5](#)**Subjects**

- A/D converter
- Timer
- GLCD

**Task**

Program a simple (egg) timer which has a START and a RESET button. Use PORTB0 for the START and PORTB1 for the RESET button. Always display the current time on the GLCD display in the "MM:SS" format where MM are the minutes and SS are the seconds. Additionally display a circle as clock face and a hand showing the remaining seconds.

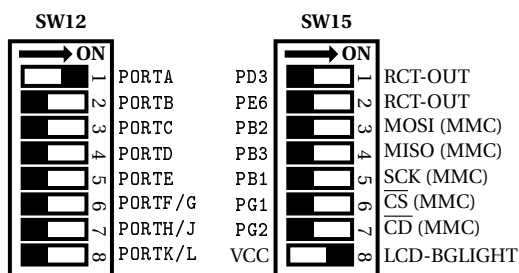
**IDLE** Initially the clock is in the IDLE state.

Now the user can configure the time using the trimmer. The possible time should be between 1 and 59 minutes.

When the user presses the START the timer changes to the COUNTDOWN state. The RESET button is ignored in this state.

**COUNTDOWN** In the COUNTDOWN state the timer counts down and refreshes the display with a frequency of 1 or 2Hz. When the RESET button is pressed the clock changes back to the IDLE state. The START button is ignored in this state.

**ALARM** In this state blink the LEDs of PORTA with 2Hz waiting for a press on the RESET button. The START button is ignored in this state.

**Pin assignment**

All other switches are set to off. Furthermore set **J15** to PF0 and **J18** to VCC.

**Remarks**

- Don't forget to use the right sleep mode.

**3.10.16 Touchscreen calibration**

★ ★

Based on: [3.8.7](#)

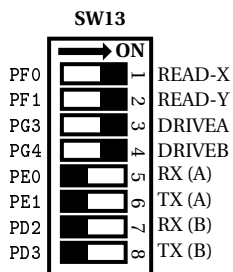
**Subjects**

- Touch panel

**Task**

Extend the touch panel driver from 3.8.7 by calibrating one or two corner points for offset and maybe scale correction. For a simple offset correction you'll only need one point and for scale correction two points. At first only implement offset correction and if the resulting values are too inaccurate add scale correction.

When programming a normal application with uses the touchscreen, start the calibration process on startup (display a cross where the user should press) and keep this configuration. The values which are given to software should be automatically corrected.

**Pin assignment**

All other switches are set to off.

**3.10.17 Graphical user interface**

★ ★

Based on: 3.10.16, 3.8.5

**Subjects**

- GLCD
- User interface
- Keypad



**Task**

Write a library for handling user input using the GLCD, the touch panel and the keypad. The user interface should support some general controls like buttons, checkboxes, optionsboxes, sliders and/or number boxes. The behaviour of the controls listed above is following:

Type	Behaviour	Datatype
Button	When touched some routine gets called exactly once.	Boolean
Checkbox	When touched it gets (de)activated.	Boolean
Optionbox	When touched, this option gets selected, others deactivated.	Integer
Slider	When touched slider changes to selected value.	Integer
Round slider	Same as slider, but round sliding wheel.	Integer
Number box	When touched it gets selected then input is possible using the keypad.	Integer
Text box	Like number box but input is read from PS/2 keyboard or UART.	String
Combo box	One value selectable from a given set of choices.	Integer

For managing the user interface following structure seems be useful: a user interface element contains a  $x_{low}$ ,  $y_{low}$ ,  $x_{high}$  and  $y_{high}$  position (outer dimensions for handling touch events). Furthermore it contains a pointer to some value which gets changed when user interacts with the element and a pointer to a string which is displayed as caption. For rendering and handling touch and keyboard input function pointers will be needed. The rendering function pointer should take the GUI element and a boolean if it's selected. The touch handler needs the GUI element, the  $x$  and  $y$  position. The keyboard handler needs again the GUI element and the character which was entered. For storing all these GUI elements just use an array with a predefined maximum amount of GUI elements. It will be also good to store the selected element to render it different if it is selected.

To manage user input and rendering right you can use a loop which first renders the user interface then waits for some user input, handles the given input and starts with rendering again.

For example a button has a value which is TRUE when it was pressed in the user handling procedure and set to FALSE if rendering is called. In the rendering position a simple rectangle is drawn from  $(x_{low}/y_{low})$  to  $(x_{high}/y_{high})$  and text is written inside and the value is set to FALSE. In the touch handler a check is made if the touch affects the button (is inside the bounding box), if so the value is set to TRUE. In the keyboard handler you can do the same as the touch handler does when  (' ') or  ('\\n') is pressed. In the user handling function in the main loop just check if the button value is set to TRUE and if so the button was pressed.

## 3.11 Games

### 3.11.1 LED dice

★★

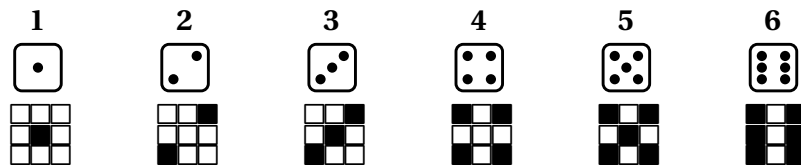
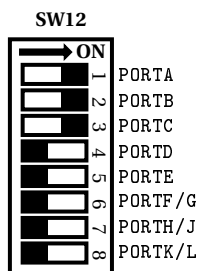
Based on: [3.10.1](#)

#### Subjects

- LEDs
- Random
- External interrupt

**Task**

Modify the random generation exercise [3.10.1](#) to generate values between 1 and 6. Display the generated number like a face of a dice on the LEDs. Use  $3 \times 3$  LEDs for the pattern.

**Pin assignment**

All other switches are set to off.

**3.11.2 Simon says**

★ ★

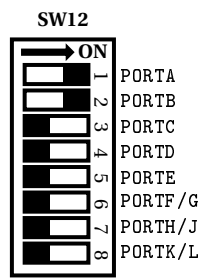
Based on: [3.10.1](#), [3.8.2](#)

**Subjects**

- LEDs
- Random
- Debouncing
- Timer

**Task**

Program the classical "Simon says" game based on the random generation exercise [3.10.1](#). You'll need a  $2 \times 2$  pattern of LEDs and buttons and button debouncing is strongly recommended. The game works the following, it starts with a sequence of 1, so at first one of four LEDs lights up, then you'll have to press the according buttons in the correct order. If the player entered the sequence with length  $n$  correct it continues with a sequence length of  $n + 1$ . For the next element generate a random value between 1–4. Always show the whole sequence (so when you have a sequence of length  $n$ ,  $n$  patterns are displayed and checked). You can assume a maximum sequence length of 64. Display every pattern for about 1s and then wait for user input.

**Pin assignment**

All other switches are set to off.

**3.11.3 Human Response Time**

★ ★

Based on: [3.8.3](#), [3.10.1](#)

**Subjects**

- LEDs
- Buttons
- Random
- LCD

**Task**

Measure the Human Response Time (HRT) using a button and a timer. At first light up a LED (PORTA0) and wait until the user presses and holds the button. When the user holds the button PORTA7 : 0 should light up. Then a random number is generated and taken as time value

$$t_{\text{WAIT}} = n_{\text{RANDOM}} \cdot 20\text{ms} + 1\text{s}$$

to wait. When the time is over PORTA7 : 0 is cleared and then the user should immediately release the button. The time between clearing the lights and releasing the button is the HRT. Display it on the LCD in ms. In the case that the user releases the button before 1s is over (doesn't hold the button) just display "Hold Button" on the LCD.

**Remarks**

- You can disable the LCD synchronization timer when measuring the HRT.

**Questions**

1. Do we need button debouncing in this task? Why will or won't we need it?



## References

- [1] MCLU Team. Boardtest, 2012. URL <http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/demo-programs/board-test/view>. [Online; accessed 05-March-2012]. 6
- [2] ATMEL. Atmega640/1280/1281/2560/2561 datasheet, 2012. URL <http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/atmega1280/atmega1280-manual/view>. [Online; accessed 05-March-2012]. 7, 9, 10, 13, 14, 16, 17, 18, 19, 20, 21, 35, 39, 41, 44, 50, 51
- [3] MCLU Team. Getting started, 2012. URL <http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/getting-started-guide/view>. [Online; accessed 05-March-2012]. 8
- [4] MCLU Team. Assembler demo, 2012. URL <http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/demo-programs/assembler-demo/view>. [Online; accessed 05-March-2012]. 9
- [5] ATMEL. Instruction set manual, 2012. URL <http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/atmega1280/avr-instruction-set/view>. [Online; accessed 05-March-2012]. 9, 10, 11, 19
- [6] MCLU Team. Makefile demo, 2012. URL <http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/demo-programs/makefile-demo/view>. [Online; accessed 05-March-2012]. 10
- [7] MCLU Team. Assembler interrupt demo, 2012. URL <http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/demo-programs/interrupt-demo/view>. [Online; accessed 05-March-2012]. 17
- [8] MCLU Team. Lcd assembler demo, 2012. URL <http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/demo-programs/lcd-assembler-demo/view>. [Online; accessed 05-March-2012]. 21, 24
- [9] MCLU Team. Calling conventions demo – server, 2012. URL <http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/demo-programs/calling-conventions-demo-ii/view>. [Online; accessed 05-March-2012]. 22
- [10] MCLU Team. Calling conventions demo – client, 2012. URL [http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/demo-programs/calling-conventions-demo-i/at\\_download/file](http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/demo-programs/calling-conventions-demo-i/at_download/file). [Online; accessed 05-March-2012]. 23
- [11] unknown. Problem 14 – project euler, unknown. URL <http://projecteuler.net/problem=14>. [Online; accessed 16-October-2011]. 25
- [12] Wikipedia. Sieve of eratosthenes — wikipedia, the free encyclopedia, 2011. URL [http://en.wikipedia.org/w/index.php?title=Sieve\\_of\\_Eratosthenes&oldid=455084540](http://en.wikipedia.org/w/index.php?title=Sieve_of_Eratosthenes&oldid=455084540). [Online; accessed 16-October-2011]. 25

- [13] MCLU Team. 4x4 matrix keypad assembler demo, 2012. URL <http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/demo-programs/4x4-matrix-keypad-assembler-demo/view>. [Online; accessed 05-March-2012]. 27
- [14] MCLU Team. C makefile demo, 2012. URL <http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/demo-programs/c-makefile-demo/view>. [Online; accessed 05-March-2012]. 28
- [15] MCLU Team. C interrupt demo, 2012. URL <http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/demo-programs/c-interrupt-demo/view>. [Online; accessed 05-March-2012]. 32
- [16] MCLU Team. C interrupt demo, 2012. URL <http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/demo-programs/c-interrupt-demo/view>. [Online; accessed 05-March-2012]. 33
- [17] avr-libc documentation. Frequently asked questions, 2012. URL [http://www.nongnu.org/avr-libc/user-manual/FAQ.html#faq\\_reg\\_usage](http://www.nongnu.org/avr-libc/user-manual/FAQ.html#faq_reg_usage). [Online; accessed 22-February-2012]. 48
- [18] MCLU Team. C lcd demo, 2012. URL <http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/demo-programs/c-lcd-demo/view>. [Online; accessed 05-March-2012]. 54
- [19] MCLU Team. C glcd demo, 2012. URL <http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/demo-programs/c-glcd-demo/view>. [Online; accessed 05-March-2012]. 56
- [20] MikroElektronika. Keypad 4x4 manual, 2012. URL [http://www.mikroe.com/eng/downloads/get/1215/keypad\\_manual\\_v100.pdf](http://www.mikroe.com/eng/downloads/get/1215/keypad_manual_v100.pdf). [Online; accessed 05-March-2012]. 61
- [21] MikroElektronika. Sht1x board user manual, 2012. URL [http://www.mikroe.com/eng/downloads/get/1547/sht1x\\_manual\\_v100.pdf](http://www.mikroe.com/eng/downloads/get/1547/sht1x_manual_v100.pdf). [Online; accessed 05-March-2012]. 61, 73
- [22] Sensirion. Datasheet sht1x, 2012. URL [http://www.sensirion.com/en/pdf/product\\_information/Datasheet-humidity-sensor-SHT1x.pdf](http://www.sensirion.com/en/pdf/product_information/Datasheet-humidity-sensor-SHT1x.pdf). [Online; accessed 05-March-2012]. 61, 72, 73
- [23] MikroElektronika. Light 2 frequency manual, 2012. URL [http://www.mikroe.com/eng/downloads/get/1477/light\\_freq2\\_manual\\_v100.pdf](http://www.mikroe.com/eng/downloads/get/1477/light_freq2_manual_v100.pdf). [Online; accessed 05-March-2012]. 62
- [24] J. Hlavá andč and, R. Ló andrencz, and M. Hadá andč andek. True random number generation on an atmel avr microcontroller. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, volume 2, pages V2–493–V2–495, april 2010. doi: 10.1109/ICCET.2010.5485568. 64

- [25] Wikipedia. Random number generation — wikipedia, the free encyclopedia, 2012. URL [http://en.wikipedia.org/w/index.php?title=Random\\_number\\_generation&oldid=475970837](http://en.wikipedia.org/w/index.php?title=Random_number_generation&oldid=475970837). [Online; accessed 18-February-2012]. 65
- [26] Wikipedia. Pseudorandom number generator — wikipedia, the free encyclopedia, 2012. URL [http://en.wikipedia.org/w/index.php?title=Pseudorandom\\_number\\_generator&oldid=472622143](http://en.wikipedia.org/w/index.php?title=Pseudorandom_number_generator&oldid=472622143). [Online; accessed 18-February-2012]. 65
- [27] dzach. Gps simulator, 2012. URL <http://sourceforge.net/projects/gpsfeed/>. [Online; accessed 05-March-2012]. 65
- [28] Wikipedia. Stevens' power law — wikipedia, the free encyclopedia, 2011. URL <http://en.wikipedia.org/w/index.php?title=Stevens1493940>. [Online; accessed 17-February-2012]. 69