HOCHSCHULE
RHEIN-WAAL
Rhine-Waal University
of Applied Sciences

LABORATORY

Microcontroller

2

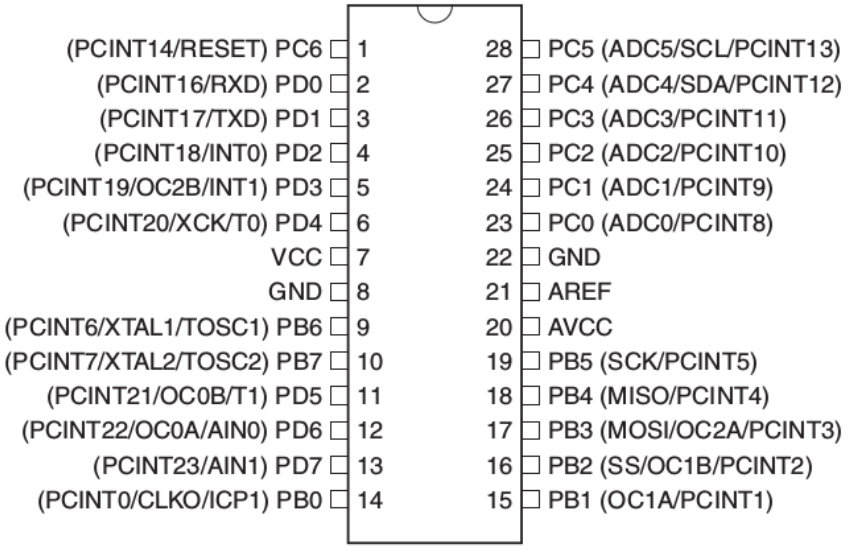
EXPERIMENT:

Timers/Counters and Interrupts

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

1 System Clock

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



External crystal oscillator is connected to pins 9 and 10 (usually together with some small capacitors).

Figure 1: Pin overview of Atmel Mega88PA Microcontroller.

2 Timers and Counters

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

Timers have a control register and also a register that holds the counter value. One of the features in the control register is the selected prescaler value. The control registers are called TCCR0A and TCCR0B (for Timer0), TCCR1A and TCCR1B (for Timer1). Here's a snippet from the datasheet for Timer1, see Figure 2 and Figure 3. The least significant bits at locations of 0 to 2 determine the prescaler value for Timer1.

Timer/Counter 1 Control Register B – TCCR1B

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

Figure 2: Desription of Timer/Counter1 Control Register B.

Table 16-5. Clock Select Bit Description

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

Figure 3: Clock Select Bit Description for Timer 1.

The TCCR0 is the 8-bit control register for Timer0. There are only 8 bits to turn on and off. TCCR1 is 16-bit, so it has more settings, but they come in two 8-bit registers labeled A and B (TCCR1A and TCCR1B). The register that holds the count value, is called the TCNT**X** register. And there is an 8-bit version (TCNT0) and a 16-bit version (TCNT1). The TCNT1 register actually consists of two 8-bit registers (TCNT1H and TCNT1L, Low and High bytes) to create a complete 16-bit number. If we want to stick to the example and turn on the Timer1 with a prescaler of 64, we would set CS11 and CS10 in the TCCR1B register to 1. This is done in the same way, like setting bits in the PORT register (without affecting the other bits), using the logic OR (bitwise OR) operator |. The counter value (which can be accessed by using "TCNT1") would then be incremented every 8 microseconds, Figure 4 explains this visually. We can not only read the counter value in TCNT**X**, but can also write (e.g. setting it back to zero or to any other value) to it.

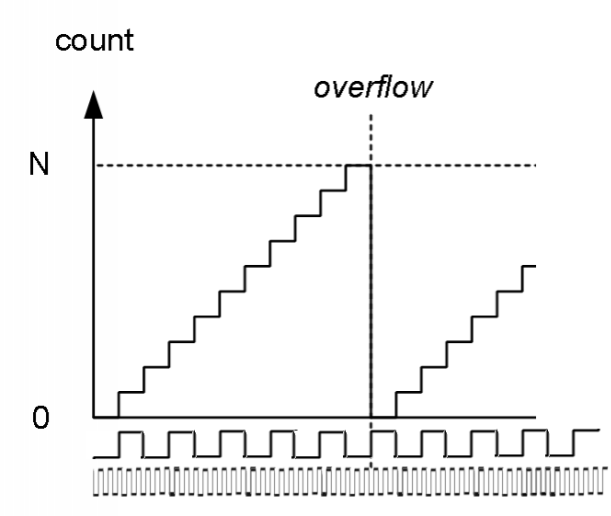


Figure 4: Function of a Timer/Counter: Overflow principle, prescaled values and the system clock.

2.1 Flashing LED

Task 1:
Use the template files (in "Templates/Experiment2"), modify the init function in such a way that Pin 0 on PortB is used as the only output. There will be no necessary input at the moment.
Connect a red LED to this Pin B0 and make it blink every second (i.e. $1Hz$ with a duty cycle of 50%) by using Timer1.

For this, you can use a simple compare (if counter value is equal or greater than... do something), to trigger the action on the outputs, like `if(XXX >= YYY) {...; XXX = 0;}`.

2.2 Audio Output

Task 2:
Generate a $3xx$ Hz signal (50% duty cycle) using the same setup (Timer1, red LED) as precisely as possible (use the smallest possible prescaler value).
For xx please use the last two digets of you matriculation number. For example with the number 12345, generate a 345 Hz signal.

Here it is necessary to change the prescaler in order to achieve better precision. The LED should now be 50% dimmer than before. You are applying a $3xx$ Hz software PWM signal (with fixed 50% duty cycle) to the LED, see Figure 5. The Atmel Mega88PA has an internal hardware to generate PWM signals, do not use this here. The LED only appears to be dimmer, because the human eye cannot recognise the LED being turned on and off very fast. Now connect the piezo buzzer instead of the LED.You should hear a $3xx$ Hz (not very clear) tone.

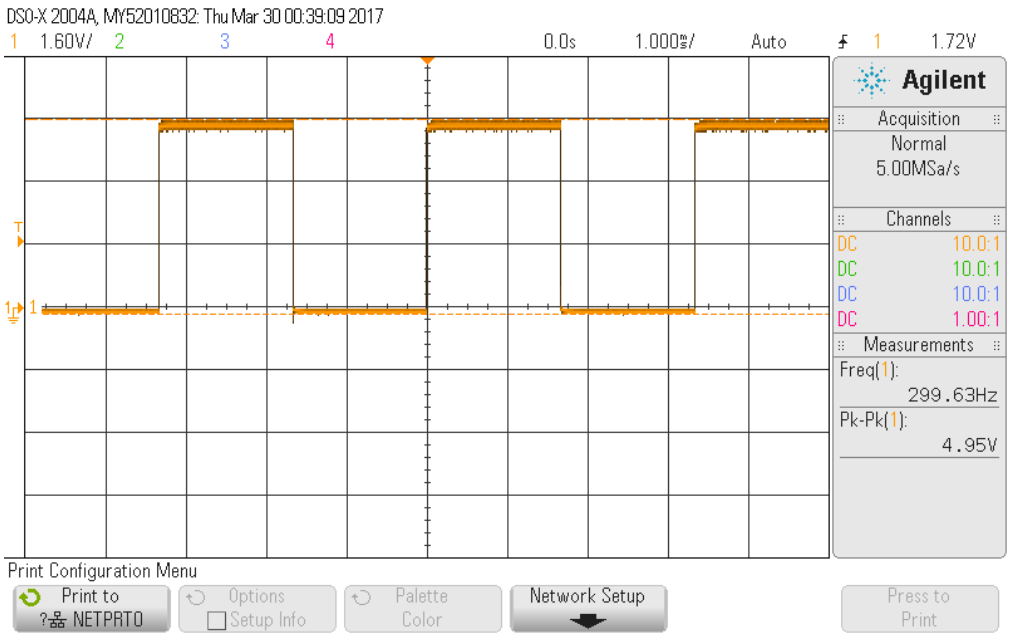


Figure 5: Screenshot of a 300 Hz signal captured with an oscilloscope.

3 Using two Timers/Counters

Task 3 (Optional):
Use two Timers/Counters to play a song. One timer is to generate the tone pitch, and the other for the tone length. **Hint:** Check the example programs from the lab 1!

The tones for ”silent night” are:

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

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

Here is a hint to make eventual changes more easy:

```
#define A 36
#define B 32
#define C 30
#define E 47
#define G 40
#define G1 20
#define END 1
```

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

Task 4 (Optional):
Connect the key 1 to the pin 1 of the MCU (called “Reset”). Press this key during the program execution. Explain what happens in your lab report.

4 Interrupt Basics

Usually, a running program is executed step by step. But for some tasks, a very fast reaction is required. To react on defined events, the program needs to ”look” for the event as often as possible. This can be a problem, as it is very inefficient and produces in general bad code, see Figure 6.

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

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

Typical interrupt events:

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

On most microcontrollers, every event has his own event handler. This handler is called **I**nterrupt **S**ervice **R**outine (ISR). The interrupt hardware stops the normal program execution and starts the ISR. After the ISR is done, the normal program will continue from the same point, see Figure 7.

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

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

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

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

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

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

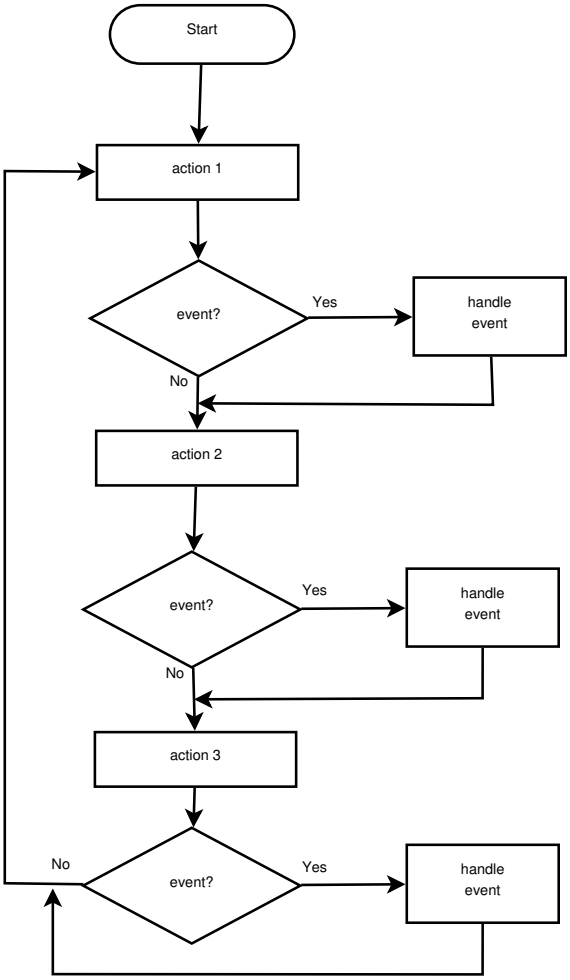


Figure 6: Program execution without interrupts.

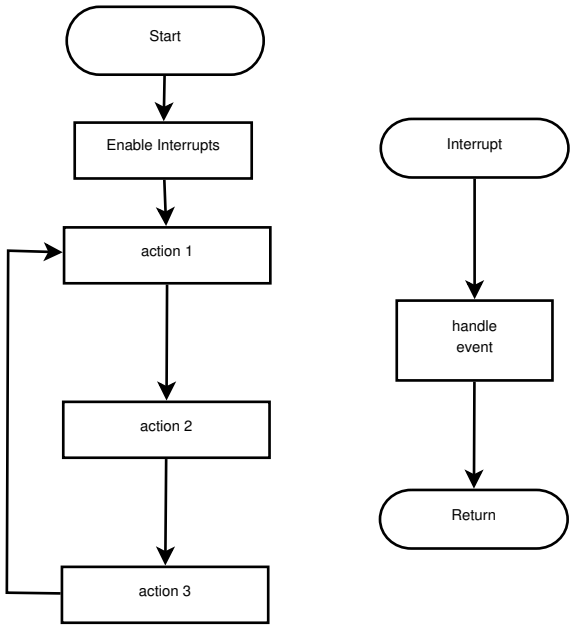


Figure 7: Better program flow with interrupts.

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

Figure 8: External Interrupt Mask Register

4.1 Interrupts on the Atmel ATmega88PA

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

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

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

Please note that the hardware executes interrupts without any security check. If an interrupt is enabled without corresponding ISR / undefined ISR, the microcontroller can hang up / crash (no hardware damage)!

The input pins PD2 and PD3 can be used for external inputs. To enable the external interrupts set the bits INT0 / INT1 in the EIMSK register, see Figure 8.

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

4.2 ISR in C

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

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

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

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

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

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

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

/* Timer/Counter1 Compare Match B */
#define TIMER1_COMPB_vect _VECTOR(7)
#define SIG_OUTPUT_COMPARE1B _VECTOR(7)

//...
```

Examples of some ISR functions in C:

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

//...

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

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

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

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

5 Using Interrupts

Before we continue with the following programs, remove the LCD display from the board and connect:

- Pin D2 to key 1,
- Pin D3 to key 2,
- Pin B0 to red LED,
- Pin B1 to yellow LED,
- Pin B2 to green LED,
- Pin C0 to potentiometer 1.

Use the template in "Template/Experiment2", please note that **the init function in this template is incomplete**.

5.1 Naive Projection

Task 5:
Write a “bad program” (without using interrupts) that:

- let the yellow LED blink with 0.5 sec delay (use _delay_ms(250) two times), and
- enables the red LED when button 1 is pressed,
- disables the red LED when button 2 is pressed.

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

During the test of this program, you can see that it has a bad reaction to the button presses. This is because the device is waiting / doing nothing during the execution of the delay function.

5.2 Digital Input Interrupts

Task 6:
To solve the problem from 5.1, it is time to interrupt the CPU in the waiting time (during delays). Move the enabling and disabling LEDs from the infinite loop / main function to the interrupt service routines INT0 and INT1. **Important:** Is the *if statement* still necessary (inside the ISR) to check if a button is pressed?

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

- set bits in register EICRA,

- set bits in register EIMSK,
- enable the global interrupt handling (already done here),
- write the code for LEDs into these two ISR functions.

Hint: The main program flow should not be influenced by button presses. An interrupt should be as fast as possible (no delays inside). If the main program will hang when a button is pressed, think about modification of the settings of EICRA.

5.3 Timer Interrupts

Task 7:
This task should be an extension of the previous task (make a copy of your previous code (whole folder) before you start here!): Use Timer/Counter1 to flash the green LED with the overflow interrupt from timer one. For the prescaler use the setting 256.

The Timer/Counter part of the Atmel Mega88PA can do more. So what if we want a flashing LED that has a short on-time and a long off-time (low duty cycle)? With the solution until now, we can just realize a 50 percent duty cycle.

So let's use the compare interrupts. These interrupts are called, if the timer/counter reaches a defined value.

Task 8:
This task should be an extension of the previous task (make a copy of your previous code (whole folder) before you start here!): Use the interrupts for the "compare register A" and the "overflow interrupt" to:

- enable the green LED when the Timer/Counter 1 overflows,
- disable the green LED when Timer/Counter 1 reaches the value 1000.

Now the green LED should have a very short on-time.

Task 9 (Optional):
Modify the last program code to use instead of INT0 and INT1 interrupts, the more general pin change interrupts PCINT0 and PCINT1, to be used **together** with the LCD module. Do show the last pressed key on the LCD add-on (first LCD line, as e.g. "Last press: key Z", Z = 1 or 2) together with the two ADC values (second LCD line, as e.g. "ACD1: XXXX, ADC2: YYYY") from BOTH potentiometers (slowly and manually adjustable).

Task 10 (Optional):
Read the value from the potentiometer (connected to ADC0) and set the prescaler of Timer/Counter 1 to one of the following values: 1, 8, 64, 256, 1024.