The goal of this session is to familiarize the reader with some peculiarities of programming microcontrollers that have a rich set of peripherals. After going through the exercises, you should be able to understand and write simple drivers, which allow you to use these peripherals efficiently. In particular we will have a detailed look at the analog digital converter (ADC) and timers.

In this session, we will avoid using library functions and operating system support as far as possible. The reason is that you should be able to really understand what is going on instead of using some black-box functionality. Clearly, this type of programming is often a bit cumbersome. But you will enjoy the comfort and convenience of an operating system that you will learn to use in the next session.

In this exercise you have to use the manual of the ATmega128 microcontroller. Use the following version, in order to ensure correct page numbers:

`$WORKSPACE/es/doc/atmel_atmega128l.pdf`

### Task 1: Analog Digital Converter (ADC)

An analog to digital converter (ADC) is used to convert a (continuous) analog signal into a (discrete) digital value. On the BTnode’s ATmega128 microcontroller this is done by comparing the voltage of the analog input signal $V_{in}$ with a reference voltage $V_{ref}$. The ATmega128’s ADC value has a 10 bit resolution and hence can digitize $2^{10} = 1024$ distinct values. If we use a single ended conversion, the result of the ADC is:

$$ADC = \lfloor 1023 \cdot \frac{V_{in}}{V_{ref}} \rfloor.$$ 

In this exercise we want to sample the battery power and show the result using the LEDs. For this we need to configure the ADC using the **ADC multiplexer select (ADMUX)** and **ADC control and status register A (ADCSRA)** configuration registers. Let us start with the ADMUX:

As you can see in the figure below, all bits are cleared at startup and we only have to set the bits to select a function required. Looking at the BTnode schematics, we see that the BAT\_SENSE signal is connected to pin 3 of port F. From the ATmega128 manual (page 239) we know that this pin is the third channel of the ADC and table 98 on page 244 tells us that we have to set the **multiplexer** bits MUX1 and MUX0 from the ADMUX register to sample the voltage from channel three. We leave the **ADC left adjust result (ADLAR)** bit cleared. The **reference selection bits (REFS1 and REFS0)** bits are left cleared because we
use the external voltage reference connected to the AREF pin of the ATmega128. **Warning:** Do not use other settings for the REFSn bits, it could destroy the microcontroller!

Altogether we only have to set MUX1 and MUX0 bits in order to select the battery voltage as the input for the ADC. For this we create a bitmask that sets the MUXn bit, by shifting the value 1 by MUXn positions: $1 << \text{MUX}_n$. We then use a logical OR operation (|) to set the bits in the ADMUX register.

```
ADMUX |= 1<<MUX0; // set bit MUX0
ADMUX |= 1<<MUX1; // set bit MUX1
```

More general, if we would like to set the bits MUX4 (=4) and MUX0 (=0) and set all other bits to zero we can use the following command.

```
ADMUX = 1<<MUX4 | 1<<MUX0; // set bit MUX4 and MUX0 only
```

If on the other hand we need to read a bit from a register we can do the following:

```
int ret = ADMUX & (1<<MUX4); // returns whether MUX4 bit is set in ADMUX
```

### Task 1.1: Configure ADC control and status register A (ADCSRA)

Now its your turn to configure the ADCSRA register. For maximal precision, we want the slowest conversion speed. We do not use interrupts and we want to do a single conversion. As the last configuration step, we want to start the conversion. Which of the bits of the ADCSRA register have to be set and in which order? How can it be determined when the conversion is completed? **Hint:** See on page 245ff. of the manual.

### Task 1.2: ADC Calculation

The 10-bit result of the conversion can be read from the ADCL and the ADCH registers. Determine the values you expect from the ADC for a battery voltage of 1 volt and 2 volts, knowing that the reference voltage is 3.3 V, the ADC delivers 10 bit values and the BAT_SENSE signal is half the battery voltage (see schematics).
Task 1.3: ADC Implementation

Write a program that shows the battery level using LEDs (red LED indicates the battery level below 1 V, yellow LED below 2 V and green LED above 2 V). Complement the following code structure.

Please make sure that two AA batteries are inserted in the BTnode and that the power switch is on. Please remind the following command line to compile and upload the code.

```
make btnode3 upload
```

```c
#include <hardware/btn-hardware.h>
define LED_BLUE 0x01
define LED_RED 0x02
define LED_ORANGE 0x04
define LED_GREEN 0x08
define LED_NONE 0x00

void write_led(u_char value)
{
    volatile u_char * pointer;
    u_char dummy;
    pointer = (u_char * ) ( ((u_short)value) << 8);
    dummy = * pointer;
    sbi(PORTB, 5);
    asm volatile ("nop" ::);
    cbi(PORTB, 5);
}

int get_battery_voltage(void) {
    int result;
    // start conversion and wait for result
    // read result
    return result;
}

int main(void) {
    int battery_voltage;
    // Configure DDRB register for using LEDs
    DDRB |= 1<<DDB5;
    // configure ADMUX
    ADMUX |= 1<<MUX0;
    ADMUX |= 1<<MUX1;
    // configure ADCSRA register such that the conversion
    // is as slow as possible and the ADC is enabled
    while (1) {
        battery_voltage = get_battery_voltage();
        // if battery_power below 1 V, switch on red LED
        // if battery_power between 1 V and 2 V, switch on yellow LED
        // if battery_power above 2 V, switch on green LED
```
// wait around a second (hint: cascade two for loops)

write_led(LED_BLUE); // write blue LED

// wait around a second

return 0;

Task 2: Interrupt Routines: Hardware Timers

If an interrupt is triggered, the normal program flow (the main function, in our case) is interrupted and the interrupt service routine (ISR) is executed. As soon as it terminates, the normal program flow is resumed exactly at the position where it was interrupted. Typical usage for interrupts are timers and peripherals such as UART or ADC.

In this exercise we learn about interrupts in the case of hardware timers. Timers are counters that are incremented automatically. When the counter reaches a certain threshold, an interrupt is triggered. Timer interrupts can thus be used to execute some periodic functionality without having to spend the whole processing time on waiting.

In this exercise we use the ATmega128’s Timer/Counter3 (manual pages 109ff.) The timer can be triggered either when the counter overflows or the counter reaches a certain threshold \(v\). This is configured with the extended Timer/Counter interrupt mask register (ETIMSK) register. As you can read in the manual (page 139ff.), the bit overflow interrupt enable \(TOIE3\) configures the timer to interrupt when the counter overflows. On the other hand, the bit output compare A match interrupt enable \(OCIE3A\) triggers based on a threshold \(v\): this (16-bit) threshold has to be set using the two registers output compare registers 3 A (OCR3AH and OCR3AL).

The speed of incrementing the timer can be adjusted: the so called prescaler value \(p\) is used to increment the counter/timer every \(p\)-th clock cycle. A small prescaler value (e.g., \(p = 1\)) provides a fine granularity of the timer, however results in frequent overflows of the counter (and therefore allows for short interrupt periods only). A large prescaler value (e.g., \(p = 1024\)) on the other hand has a coarse granularity, but allows for longer interrupt periods. The timer/counter 3 is configured using the timer/counter3 control register B (TCCR3B). In particular the clock select registers (first three bits CS3n) are used to configure the prescaler (see page 135 in the manual).

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETIMSK</td>
<td>-</td>
<td>-</td>
<td>TICIE3</td>
<td>OCIE3A</td>
<td>OCIE3B</td>
<td>TOIE3</td>
<td>OCIE3C</td>
<td>OCIE1C</td>
</tr>
<tr>
<td>Read/Write</td>
<td>R</td>
<td>R</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>Initial Value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCCR3B</td>
<td>-</td>
<td>-</td>
<td>WGM33</td>
<td>WGM32</td>
<td>CS32</td>
<td>CS31</td>
<td>CS30</td>
<td></td>
</tr>
<tr>
<td>Read/Write</td>
<td>R/W</td>
<td>R/W</td>
<td>-</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td></td>
</tr>
<tr>
<td>Initial Value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Task 2.1: Configure Prescaler

How do we need to configure the prescaler if we need the timer to interrupt every 2 seconds? To what threshold \(v\) do we need to set the timer? **Hints:** The clock frequency is 7.3 MHz. The counter is 16-bit. This is a purely theoretic exercise.
Task 2.2: Understanding an Interrupt Implementation

In order to register an interrupt we use the low-level OS routine \texttt{NutRegisterIrqHandler(irq, handler, arg)}, which has three arguments:

- \textbf{IRQ}: Interrupt number to be associated with this handler. For instance, \texttt{sig\_OVERFLOW3} is used to register an overflow of counter 3, and \texttt{sig\_OUTPUT\_COMPARE3A} is used to trigger an interrupt if the counter reaches the threshold saved in the output compare registers (\texttt{CR3AH} and \texttt{CR3AL}).
- \textbf{Handler}: This routine is called by the OS, when the specified interrupt occurs.
- \textbf{Arg}: Argument to be passed to the interrupt handler.

Understand the following pseudo code showing a simple usage of the timer using LEDs. The program toggles the blue led every second and sets the green LED whenever counter 3 overflows.

```c
#include <hardware/btn-hardware.h>
#include <dev/irqreg.h>

static void timer3IRQ(void * arg) {
    // switch on green LED
}

int main(void) {
    // register interrupt service routine
    NutRegisterIrqHandler(sig\_OVERFLOW3, timer3IRQ, 0);
    // configure the speed of the timer (the prescaler is set to 1024)
    TCCR3B |= 1<< CS30;
    TCCR3B |= 1<< CS32;
    // enable the interrupt at overflows of the timer
    ETIMSK |= 1<< TOIE3;

    while (1) {
        // toggle the blue LED and switch off green LED
        // wait approximately a second (using a for loop)
    }
    return 0;
}
```

Task 2.3: Implement ADC with Interrupts

Modify the program from Task 1.3 (ADC) such that the battery level is sampled every 2 seconds using a timer interrupt. You may use the template below.

\textit{Optional}: Also use an interrupt for signalling when the result of the ADC is ready. For this you to register a second interrupt and you have to adapt the configuration of the ADC (register \texttt{ADCSRA}). \textbf{Hint}: The IRQ number associated with the ADC interrupt is \texttt{sig\_ADC}.

```c
#include <hardware/btn-hardware.h>
#include <dev/irqreg.h>

// this function is triggered when the timer expires
static void timer3IRQ(void * arg)
{
    // configure ADC
    // start conversion and wait for result
    // indicate battery status using LED
}
```
int main(void)
{
    int toggle = 0;

    // Configure DDRB register for using LEDs
    DDRB |= 1<<DDB5;

    // Register timer3 interrupt using compare register
    NutRegisterIrqHandler(&sig_OUTPUT_COMPARE3A, timer3IRQ, 0);

    // Configure timer3
    // The timer should expire exactly every 2 seconds
    // Reset timer when threshold reached
    TCCR3B |= 1<<WGM32;
    // start timer3

    while (1) {
        if (toggle) {
            toggle = 0;
            // switch on blue LED
        } else {
            toggle = 1;
            // switch off blue LED
        }
        // pause for about one second (use for loop)
    }

    return 0;
}