

## Embedded Systems FS 2012

### Solution to Lab 2: Peripherals, Interrupt Service Requests

Discussion Date: 21. March 2012

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:

[http://www.btnode.ethz.ch/pub/uploads/Documentation/atmel\\_atmega1281.pdf](http://www.btnode.ethz.ch/pub/uploads/Documentation/atmel_atmega1281.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:  $ADC = \lfloor 1023 \cdot V_{in}/V_{ref} \rfloor$ .

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

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

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:  $1 \ll \text{MUXn}$ . 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.

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

**Solution** For the highest accuracy (slowest speed), we set the bits *ADC prescaler select bits* (ADPS0, ADPS1 and ADPS2) to 1. We leave the *ADC interrupt flag* (ADIF) and *ADC interrupt enable* (ADIE) zero since we do not use interrupts. We also leave the *ADC free running select* (ADFR) zero, since we only perform one single conversion. We set the *ADC enable* (ADEN) to 1 to enable the ADC and set the *ADC start conversion* (ADSC) to 1 to start the conversion. As soon as the ADSC is set back to 0, the conversion is completed.

```
ADCSRA |= 1<<ADPS0;
ADCSRA |= 1<<ADPS1;
ADCSRA |= 1<<ADPS2;
ADCSRA |= 1<<ADEN;
ADCSRA |= 1<<ADSC;
```

## 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).

**Solution** If battery voltage is 1 V, the BAT\_SENSE signal is 0.5 V.  $ADC_{1V} = \lfloor 1023 \cdot 0.5V/3.3V \rfloor = 155$ . For a battery voltage of 2 V, ADC value is 310.

Bit	15	14	13	12	11	10	9	8	
	-	-	-	-	-	-	ADC9	ADC8	ADCH
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	

## 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
```

```
adc_polling_solution.c
```

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

// Compute ADC values corresponding to 1 and 2 Volts:
// If battery voltage is 1V, the BAT_SENSE signal is 0.5V.
// The reference voltage is 3.3V and corresponds to the ADC value 1023.
// Therefore 0.5V corresponds to an ADC value of 1023/3.3*0.5=155.
// A battery voltage is 2V, ADC value is 310.
#define BAT_1_VOLT 155
#define BAT_2_VOLT 310

void pause(u_short duration)
{
    u_short i;
    u_short ii;
    for (i=0;i<duration;i++) {
        for (ii=0;ii<0xffff;ii++) {
        }
    }
}

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 ADC conversion
    ADCSRA |= 1<<ADSC;

    // Poll ADCSRA register (bit ADSC) to check whether conversion is finished
    while (ADCSRA & (1<<ADSC)) ;

    // Read 10 bit results from ADCL and ADCH
    result = ADCL;
    result |= ADCH << 8;

    return result;
}

int main(void)
{
    int battery_voltage = 0;

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

    // Configure ADC to sample battery voltage
    ADMUX |= 1<<MUX0;
    ADMUX |= 1<<MUX1;

    // Configure ADC for maximal accuracy
    ADCSRA |= 1<<ADPS0;
    ADCSRA |= 1<<ADPS1;
    ADCSRA |= 1<<ADPS2;

    // Enable ADC
    ADCSRA |= 1<<ADEN;

    while (1) {
        battery_voltage = get_battery_voltage();
        if (battery_voltage < BAT_1_VOLT) {
            write_led(LED_RED); // write red LED
        }
        else if (battery_voltage < BAT_2_VOLT) {
            write_led(LED_ORANGE); // write orange LED
        }
        else {
            write_led(LED_GREEN); // write green LED
        }
        pause(10);
        write_led(LED_BLUE); // write blue LED
        pause(10);
    }

    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).

Bit	7	6	5	4	3	2	1	0	
	-	-	TICIE3	OCIE3A	OCIE3B	TOIE3	OCIE3C	OCIE1C	ETIMSK
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

  

Bit	7	6	5	4	3	2	1	0	
	ICNC3	ICES3	-	WGM33	WGM32	CS32	CS31	CS30	TCCR3B
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	

## 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.

**Solution** The clock frequency is 7.3 MHz, and hence the clock period is  $clk_{I/O} = 1/7.3e6 \sim 1.37e-7$  s. If the prescaler is set to  $p = 1$ , the timer overflows every  $2^{16} = 65,536$  clock periods, which corresponds to 9 ms. In order to cover a period of 2 seconds, we need a prescaler value  $p \geq 2 \text{ s} / 9 \text{ ms} = 222.2$ . According to Table 62 on page 135 in the manual, the next higher prescaler value is 256, which requires to set CSn2 to 1 and leaving CSn1 and CSn0 0.

```
TCCR3B |= 1<<CS32;
```

Setting the prescaler to 256, requires to set the threshold to  $\frac{7.3 \text{ MHz}}{256} \cdot 2 \text{ s} = 57031 = 0xdec7$ .

```
CR3AH = 0xde;
```

```
CR3AL = 0xc7;
```

```
ETIMSK |= 1<<OCIE3A; // interrupt is based the threshold above
```

## Task 2.2: Understanding an Interrupt Implementation

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

- **IRQ:** Interrupt number to be associated with this handler. For instance, `sig_OVERFLOW3` is used to register an overflow of counter 3, and `sig_OUTPUT_COMPARE3A` is used to trigger an interrupt if the counter reaches the threshold saved in the output compare registers (`CR3AH` and `CR3AL`).
- **Handler:** This routine is called by the OS, when the specified interrupt occurs.
- **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.

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

void pause(u_short duration)
{
    u_short i;
    u_short ii;
    for (i=0;i<duration;i++) {
        for (ii=0;ii<0xffff;ii++) {
        }
    }
}

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);
}

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

int main(void) {
    int toggle;

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

    // register interrupt service routine
    NutRegisterIrqHandler(&sig_OVERFLOW3, timer3IRQ, 0);
    // configure the speed of the timer
    TCCR3B |= 1<< CS30;
    TCCR3B |= 1<< CS32;
    // enable the interrupt at overflows of the timer
}
```

```

ETIMSK |= 1<< TOIE3;

while (1) {
    if (toggle) {
        toggle = 0;
        write_led(LED_BLUE);
    }
    else {
        toggle = 1;
        write_led(LED_NONE);
    }
    pause(10);
}
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.

*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 ADCSRA). **Hint:** The IRQ number associated with the ADC interrupt is `sig_ADC`.

```

adc_interrupt_solution.c

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

// Compute ADC values corresponding to 1 and 2 Volts:
// If battery voltage is 1V, the BAT_SENSE signal is 0.5V.
// The reference voltage is 3.3V and corresponds to the ADC value 1023.
// Therefore 0.5V corresponds to an ADC value of 1023/3.3*0.5=155.
// A battery voltage is 2V, ADC value is 310.
#define BAT_1_VOLT 155
#define BAT_2_VOLT 310

void pause(u_short duration)
{
    u_short i;
    u_short ii;
    for (i=0;i<duration;i++) {
        for (ii=0;ii<0xffff;ii++) {
        }
    }
}

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);
}

```

```

// This function is triggered when the ADC finished the conversion
static void readADC(void *arg)
{
    int battery_voltage;

    battery_voltage = ADCL;
    battery_voltage |= ADCH << 8;

    if (battery_voltage < BAT_1_VOLT) {
        write_led(LED_RED); // write red LED
    }
    else if (battery_voltage < BAT_2_VOLT) {
        write_led(LED_ORANGE); // write orange LED
    }
    else {
        write_led(LED_GREEN); // write green LED
    }
}

// this function is triggered when the timer expires
static void timer3IRQ(void *arg)
{
    ADCSRA |= 1<<ADSC; // Start ADC conversion
}

int main(void)
{
    int toggle = 0;

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

    // Register ADC interrupt handler
    NutRegisterIrqHandler(&sig_ADC, readADC, 0);

    // Configure ADC to sample battery voltage
    ADMUX |= 1<<MUX0;
    ADMUX |= 1<<MUX1;

    // Configure ADC for maximal accuracy
    ADCSRA |= 1<<ADPS0;
    ADCSRA |= 1<<ADPS1;
    ADCSRA |= 1<<ADPS2;

    // Enable ADC interrupt mode
    ADCSRA |= 1<<ADIE;
    // Enable ADC
    ADCSRA |= 1<<ADEN;

    //Register timer interrupt
    NutRegisterIrqHandler(&sig_OUTPUT_COMPARE3A, timer3IRQ, 0);

    // set prescaler to 256 (overflow every 2.3 s)
    TCCR3B |= 1<<CS32;

    // To get an interrupt every 2 s, the interrupt should be triggered
    // when the counter reaches 7.3MHz / 256 * 2s = 57031 = 0xdec7
    OCR3AH = 0xde;
    OCR3AL = 0xc7;

    // Enable timer interrupt
    ETIMSK |= 1<<OCIE3A;

    while (1) {
        if (toggle) {
            toggle = 0;
            write_led(LED_BLUE);
        }
        else {

```

```
        toggle = 1;
        write_led(LED_NONE);
    }
    pause(10);
}

return 0;
}
```