Interrupts and Timer

May 19th, 2020

This exercise is not graded and has to be finished within two weeks! Even though this exercise is not graded, the created library is part of the later graded submission, so coding style is important!

During the last exercise, you learned how to use buttons, the ADC, and delays to write simple applications. However, polling the status of peripherals and using blocking delay loops result in very unsatisfying timing as well as intertwined code. As a solution, the AVR microcontroller provides interrupts. That is, under certain conditions, such as pressing a button, the current program flow is stopped and a dedicated interrupt service routine (ISR) is executed before the normal program flow is continued.

In this exercise, you will learn how to use interrupts for buttons and about a more elegant way for implementing timed actions. First, create a new project for the following tasks in your workspace (see task 2.2).

Furthermore, download and extract the *sheet3_templates.zip* archive from the studIP page to the project of your SES library.

Task 3.1 : May I Interrupt you?

All buttons of the SES board trigger a single interrupt, which is called pin change interrupt. To understand this functionality, please open the ATmega128RFA1 datasheet (provided via studIP) and read section *16 External Interrupts* carefully. Furthermore, C function pointers are used in this exercise. If you need more information about function pointers, please ask Google, Bing, or any source of your personal trust.

Extend the header *ses_buttons.h* with the following three lines

```
typedef void (*pButtonCallback)();
void button_setRotaryButtonCallback(pButtonCallback callback);
void button_setJoystickButtonCallback(pButtonCallback callback);
```

Open the library file *ses_buttons.c*. Extend the `button_init()` function that you wrote in the previous exercise with the following actions:

- To activate the pin change interrupt, write a `1` to the corresponding pin in the `PCICR` register.
- To enable triggering an interrupt if a button is pressed, write a `1` to the corresponding position in the mask register `PCMSK0` (position is the same as given in `BUTTON_ROTARY_PIN` and `BUTTON_JOYSTICK_PIN` definition).

Implement the interrupt service routine `ISR(PCINT0_vect)` in the source file *ses_buttons.c*:

```
ISR(PCINT0_vect){
    // execute callbacks here
}
```

- Check whether one of the button values changed.
- Execute the appropriate button callback function.
- Make sure that a button callback is only executed if a valid callback was set and the mask register contains a `1`.

Now, implement the setter functions

- `button_setRotaryButtonCallback`
- `button_setJoystickButtonCallback`

and store the given function pointers in variables that can be accessed by the ISR. Document their usage in the header.

Having implemented our little button library, we actually want to test it. For this purpose, in the project for this task, write a program that first initializes the buttons and defines two functions for toggling the leds. Enable the buttons and pass the function pointers of the led toggle to the buttons.

When writing public libraries like the button driver, always consider that invalid parameters might be passed (such as null pointers)!

Function pointers should not be accessible outside module (static) and must not be touched by GCC optimization (volatile).

Don't forget to globally enable all interrupts by using the `sei()` function!

Don't forget the infinite *while* loop after the initialization!

Although the buttons are rather "good", depending on the board you may observe mechanical bouncing effects,e. g., the micro controller calls the interrupt service routine two times for a single button press. You may ignore these for this task.

Don't forget the delay between the camera and web interface, when you use the remote lab.

Task 3.2 : Setting up a Hardware Timer

Instead of busy-waiting with the `_delay_ms` function, the timing should be done without blocking other operations. The ATMEL ATmega128RFA1 has two 8-bit timers and four 16-bit timers. A timer is simply a special register in the microcontroller that can be incremented or decremented in hardware (e.g. by the clock signal) independently from other operations. The crucial benefit of timers is that they can trigger timer-interrupts once certain conditions are met. For instance, a timer can increment an 8-bit register and upon the transition from 255 to 0 an overflow interrupt flag is set.

For that timer at 16 MHz clock frequency 62500 interrupts are triggered per second resulting in a time of 16 µs between two interrupts. The time can be extended by using prescalers, dividing the frequency of the clock signal by a fixed number. With a prescaler of, e. g., 16 the time between two interrupts increases to 256 µs. The time can be further extended by using a larger prescaler, using 16-bit timers or implementing a software counter based on the timer-interrupts. The set of prescalers is limited depending on the microcontroller and the timer used.

In this part of the exercise we will use the 8-bit Timer/Counter2. It will be necessary to read parts of the ATmega128RFA1 datasheet chapter *21 8-bit Timer/Counter2 with PWM and Asynchronous Operation*. Especially section *21.11 Register Description* contains the most important information!

Open the files *ses_timer.h* and *ses_timer.c* and implement the given functions according to the following information:

- Use Timer/Counter2.
- Use *Clear Timer on Compare Match (CTC)* mode operation (section *21.5: Modes of Operation*).
- Select a prescaler of 64.

- Set interrupt mask register for *Compare A*.
- Clear the interrupt flag by setting a 1 in flag register for *Compare A*.
- Complete and use the macros given in *ses_timer.c*.
- Set a value in register *OCR2A* in order to generate an interrupt every 1 ms.

Do not use *magic numbers* in your code, instead use macros from the datasheet!

Implement the interrupt service routine. Only the callback function has to be invoked which was passed to `timer2_setCallback()` .

Now test your timer implementation by extending your `main()` function as well as providing a function `void softwareTimer(void)`, which will serve as callback. In `main()`, initialize the timer and pass the pointer to the function `softwareTimer()` to it. The function `softwareTimer()` should toggle the yellow LED. Since the timer is fired each millisecond you will probably need some software counter to decrease the frequency. Toggle the LED every second.

Task 3.3 : Button debouncing

In the first task, we directly triggered an interrupt on an edge in the signal caused by a button press. If you were lucky, your board had a very “good” button and you did not notice any bouncing effects. Normally, however, mechanical buttons exhibit some bouncing behavior (see Fig. 1) and directly triggering interrupts on non-debounced buttons is considered bad practice and may likely lead to unexpected behavior. Thus, instead of using polling in a main loop or using direct interrupts which may lead to multiple button presses due to bouncing, we now steer middle ground by polling the button state within a timer interrupt.

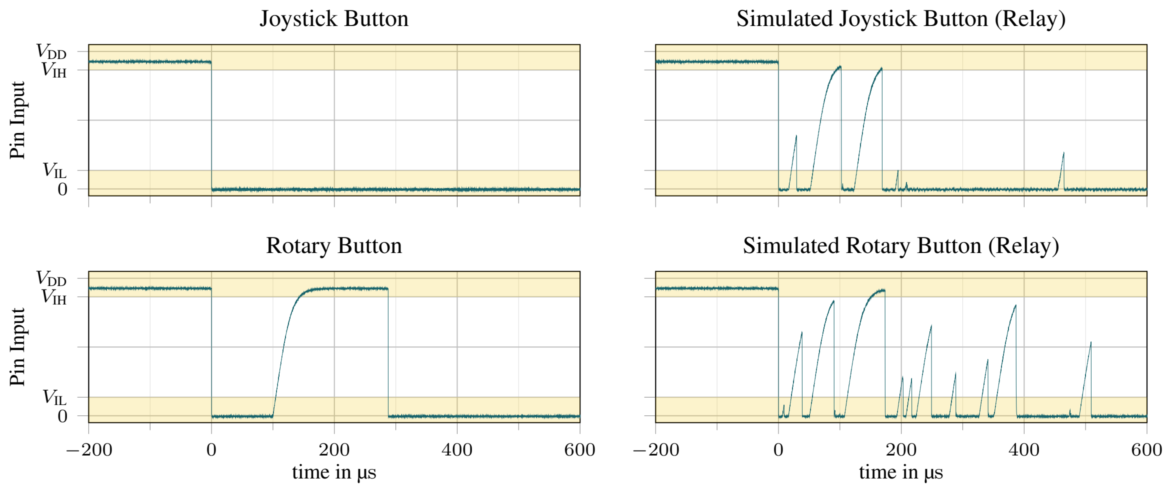

Figure 1: Measured voltages at the input pins (with activated internal pullup resistors). At first, the buttons were directly pressed. In the second case, the buttons were simulated with the help of the remote lab. The remote lab uses relays, which have also bouncing effects. V_{DD} is the supply voltage. V_{IL}, V_{IH} are the low and high level input voltage (see section *34.1.2 Digital Pin Characteristics* in the datasheet).

Implement the empty functions for Timer/Counter1:

- Achieve an interval of 5 ms.
- Make sure to select the correct mode (value differs from Timer/Counter2)!
- Use your test program to verify that the timer is configured correctly, too.

Now copy the following defines and (incomplete) function to your `ses_button.c`:

```
#define BUTTON_NUM_DEBOUNCE_CHECKS     // TODO
#define BUTTON_DEBOUNCE_PERIOD         // TODO
#define BUTTON_DEBOUNCE_POS_JOYSTICK   0x01
#define BUTTON_DEBOUNCE_POS_ROTARY     0x02
```

```
void button_checkState() {
    static uint8_t state[BUTTON_NUM_DEBOUNCE_CHECKS] = {};
    static uint8_t index = 0;
    static uint8_t debouncedState = 0;
    uint8_t lastDebouncedState = debouncedState;

    // each bit in every state byte represents one button
    state[index] = 0;
    if(button_isJoystickPressed()) {
        state[index] |= BUTTON_DEBOUNCE_POS_JOYSTICK;
    }
    if(button_isRotaryPressed()) {
        state[index] |= BUTTON_DEBOUNCE_POS_ROTARY;
    }

    index++;
    if (index == BUTTON_NUM_DEBOUNCE_CHECKS) {
        index = 0;
    }

    // init compare value and compare with ALL reads, only if
    // we read BUTTON_NUM_DEBOUNCE_CHECKS consistent "1's" in the state
    // array, the button at this position is considered pressed
    uint8_t j = 0xFF;
    for(uint8_t i = 0; i < BUTTON_NUM_DEBOUNCE_CHECKS; i++) {
        j = j & state[i];
    }
    debouncedState = j;

    // TODO extend function
}
```

Try to understand the purpose of the function and define the macro `BUTTON_NUM_DEBOUNCE_CHECKS` in your button driver. The function is meant to be called by the timer interrupt you just implemented, so it should be set as callback during the initialization.

Extend the `button_checkState` function, so that the corresponding button callback is called as soon as the debouncedState indicates a (debounced) button **press** (not release). Note, that though we are using it for two buttons only, this method could be used to debounce up to eight buttons in an efficient way.

A good value for `BUTTON_NUM_DEBOUNCE_CHECKS` and our buttons is 5, which introduces a delay of up to 30 ms. This is fast enough to feel instantaneous for a human and provides reliable debouncing. Note, however, that individual buttons may exhibit different behavior. Some nice reading is “The Art of Designing Embedded Systems” by Jack Ganssle, where the idea for the algorithm originates from.

Extend the `button_init` function so that it takes a flag that specifies if the debouncing or the direct interrupt technique should be used:

```
void button_init(bool debouncing) {
    // TODO initialization for both techniques (e.g. setting up the DDR register)

    if(debouncing) {
        // TODO initialization for debouncing
        timer1_setCallback(button_checkState);
    }
    else {
        // TODO initialization for direct interrupts (e.g. setting up the PCICR register)
    }
}
```

Test your modified button driver with the main project from the first task! What are the improvements between task 3.1 and 3.3?

Task 3.4 : The SEI Instruction (Challenge)

If you have no time left to finish this subtask during the current two weeks, please skip it to catch up in the following week.

The datasheet (section *7.8 Reset and Interrupt Handling*) states

> When using the SEI instruction to enable interrupts, the instruction following SEI will be executed before any pending interrupts, [...].

Can you think of situations where this leads to side effects or race conditions?

Find out if this is an inherent property of the SEI instruction or if this is also valid for other ways of enabling the global interrupt enable bit.

Most C instructions result in more than one microcontroller instruction. For investigating such statements, better use (inline) assembly routines. In the following, some useful routines are given.

The sei instruction in inline assembly:

```
asm volatile ("sei");
```

An alternative way of setting the SEI bit:

```
asm volatile ("in r16,0x3f \t\n\
               ori r16,128 \t\n\
               out 0x3f,r16");
```

Enabling the yellow LED in a single instruction (after initialization):

```
asm volatile ("cbi 0x11, 7");
```

What happens if the next instruction after SEI is CLI?

```
asm volatile ("cli");
```

To implement a test setup for the SEI instruction, you can follow these steps:

1. Initialize the LEDs.
2. Implement an interrupt service routine for a timer, which turns the red LED on and wait forever afterwards.
3. Start the timer (without executing SEI afterwards).
4. Waits for the timer (`_delay_ms()`).

Afterwards you can test different combinations with the SEI instruction, e. g.,

```
    asm volatile ("sei");
    asm volatile ("cbi 0x11, 7"); // yellow LED
```

or

```
    asm volatile ("sei");
    asm volatile ("nop");
    asm volatile ("cbi 0x11, 7"); // yellow LED
```