Interrupts and Timer

May 7th, 2025

In exercise two, you learned how to use buttons, the ADC, and delays to write simple applications. However, polling the peripherals' status and blocking delay loops results in unsatisfying timing and intertwined code. Instead, using the microcontroller's interrupts is a better approach for reacting to events. Under certain conditions, e.g., pressing a button, the microcontroller halts the current program flow and executes a dedicated interrupt service routine (ISR) before continuing the normal program flow.

In this exercise, you learn how to use interrupts for buttons instead of polling, and we show you a way to implement timed actions more elegantly than with the delay function.
First, create a new project for the following tasks in your workspace (cf. task 2.2).
Furthermore, download and extract the *sheet3_templates.zip* archive from Stud.IP and add the content to your SES library folder within your project.

Grading

This exercise is not graded. You will be working on it for two weeks. Even though we don't grade this exercise, you are creating a library that is part of the later graded submissions, so coding style and proper function are essential!

Task 3.1: May I Interrupt You?

All buttons of the SES board trigger the same interrupt called *pin-change interrupt*. To understand how that works in detail, please open the ATmega32U4 datasheet (provided via Stud.IP) and read section *11. External Interrupts* carefully. Furthermore, this exercise uses C function pointers, so this is an excellent opportunity to brush up on pointer skills. If you need more information about function pointers, please refer to the *ppi-handout.pdf* in Stud.IP.

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

```c
typedef void (*pButtonCallback)(void);
void button_setRotaryButtonCallback(pButtonCallback callback);
void button_setPushButtonCallback(pButtonCallback callback);
```

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

- To activate the pin change interrupt, write a 1 to the corresponding bit in the `PCICR` register.
- To enable triggering an interrupt if a button is pressed, write a 1 to the corresponding bit in the mask register `PCMSK0` (bit positions are the same as given in `BUTTON_ROTARYBUTTON_BIT` and `BUTTON_PUSHBUTTON_BIT` definitions).

Implement the interrupt service routine for `PCINT0_vect` in the source file *ses_button.c*:

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

In the interrupt service routine:

- Check if and which button states changed.
- Check if corresponding interrupt is unmasked and if callback is set.
- Only then execute the appropriate button callback function(s).

Next, implement the callback setter functions

- `button_setRotaryButtonCallback()`
- `button_setPushButtonCallback()`

Moreover, store the function pointers given in the argument in variables accessible to the ISR. Document the usage of the callback setters in the header.

After implementing our little button library, we naturally test it. For this purpose, write a program in this task's project that initializes the buttons and defines two functions for toggling the LEDs. Enable the buttons and install LED toggle functions of your choice as button callbacks.

Coding Style

- 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 the module (static) and must not be touched by GCC optimization (volatile).

Common Pitfalls

- Don't forget to globally enable all interrupts by using `sei()`!
- Don't forget the infinite **`while`** loop after the initialization in `main()`!
- Although the buttons switch relatively reliable, depending on the board, you may observe mechanical bouncing effects, i. e., a single button press causes several interrupts. You may ignore these for this task.

Task 3.2: Setting up a Hardware Timer

Instead of busy-waiting with the function `_delay_ms()`, the timing should be done without blocking other operations. This can be realized using Timers. The ATmega32U4 has one 8-bit (Timer0), two 16-bit (Timer1, 3), and one 10-bit (Timer4) timer/counter. A timer is a special microcontroller register that can be incremented or decremented automatically on a trigger (e.g., by the clock signal) independently from other operations. Timers can also trigger interrupts once certain conditions are met. For instance, a timer can increment an 8-bit register on every clock cycle, and, upon the transition from 255 to 0, it triggers an overflow interrupt.

Running such a timer at 16 MHz clock frequency will trigger 62 500 interrupts per second, resulting in a time of 16 µs between two consecutive interrupts. This 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. To further extend this interval we can use a larger prescaler, or a 16-bit timer, or we can implement a software counter based on the timer interrupts. However, the set of prescalers is limited depending on the microcontroller and the used timer.

In this part of the exercise, we use the 8-bit Timer/Counter0, so you need to read chapter 13., *8-bit Timer/Counter0 with PWM* of the ATmega32U4 datasheet. In particular, pay attention to section *13.8. 8-bit Timer/Counter Register Description*.

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

- Use Timer/Counter0.
- Use *Clear Timer on Compare Match (CTC)* mode operation (section *13.6. Modes of Operation*).
- Select a prescaler of 64.
- Set interrupt mask register for *Compare A*.
- Clear the interrupt flag by setting a 1 in the flag register for *Compare A*.
- Complete and use the macros provided in *ses_timer.c*.
- Set the register `OCR0A` in order to generate an interrupt every 1 ms. Use the macro `F_CPU` to provide a general calculation.

Coding Style

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

Implement the interrupt service routine, which only has to call the callback function installed via `timer0_setCallback()`.

Now test your timer implementation by extending your `main()` function as well as providing a function `void softwareTimer(void)`, which will serve as a callback. In `main()`, initialize the timer and install `softwareTimer()` as the callback for the timer. The function `softwareTimer()` should toggle the yellow LED. Since the function `softwareTimer()` is called every millisecond you need an additional mechanism (a timer/counter implemented in software) to decrease the frequency and toggle the LED every second.

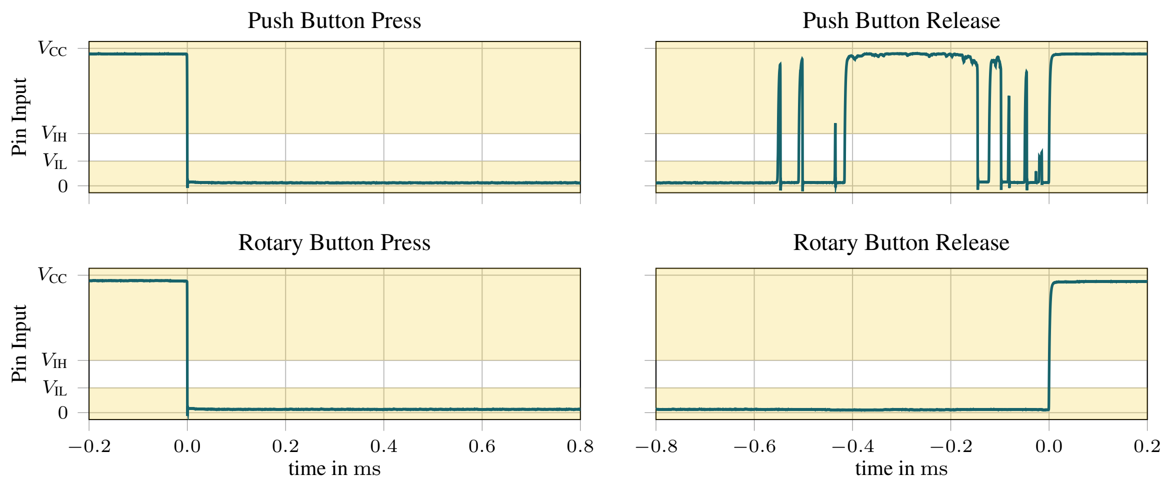

Figure 1: Measured voltages at the microcontroller pins (with activated internal pullup resistors) connected to the SES board buttons. V_{CC} is the supply voltage. V_{IL}, V_{IH} are the low and high level input voltages (see section *29.2. DC Characteristics* in the datasheet).

Task 3.3 : Button Debouncing

In the first task we trigger interrupts by a rising or falling edge of the pin voltage, caused by a button press. If you were lucky, your board had a very "good" button and you did not notice any bouncing effects. Mechanical buttons usually exhibit bouncing behavior (see Fig. 1), hence, directly using edge-triggered interrupts on non-debounced buttons is considered bad practice and may lead to unexpected behavior. Better than using interrupts directly—which may lead to ghost button presses due to bouncing—is polling the button state via a timer interrupt.

For this purpose, we use Timer/Counter1. Implement its initialization in *ses_timer.c*. Choose the correct mode (value differs from Timer/Counter0!) to achieve an interval of 5 ms. Use your test program from the previous task to verify that the timer is configured correctly.
Next, copy the defines and (incomplete) function given in the appendix "Debouncing with a ring buffer" to the already existing button library file *ses_button.c*.

The timer must call the function `button_checkState()` periodically, therefore, we set it as Timer1 callback during initialization. The function reads the states of all buttons and stores these states in the array `state`. All buttons are assigned one bit per element in the array, and the array memorizes the last `BUTTON_NUM_DEBOUNCE_CHECKS` button states in a round-robin or ring-buffer fashion; i. e., the elements of the array keep track of the history of previous buttons states. Note that though we use it for only two buttons, this method can efficiently debounce up to eight buttons.

Extend the function `button_checkState()` so that it calls the corresponding button callback as soon as the `debouncedState` indicates a (debounced) button **press** (not release). Ensure each callback is only called once per button press by using `lastDebouncedState`.

How many checks should I make?

A good value for `BUTTON_NUM_DEBOUNCE_CHECKS` and our buttons is 5, which introduces a delay of up to 25 ms. This is fast enough to feel instantaneous for a human and provides reliable debouncing at the same time. Note that individual buttons may exhibit different behavior. Some recommended reading is "The Art of Designing Embedded Systems" by Jack Ganssle, from where the algorithm's idea originates.

Finally, extend the function `button_init()`, so that it accepts a parameter (boolean flag) that specifies if the debouncing or the direct interrupt technique should be used:

```c
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 of this sheet! What are the improvements over the solution in task 3.1?

Debouncing with a ring buffer

```c
#define BUTTON_NUM_DEBOUNCE_CHECKS        /* TODO */
#define BUTTON_DEBOUNCE_POS_PUSHBUTTON    0x01
#define BUTTON_DEBOUNCE_POS_ROTARYBUTTON  0x02

...

void button_checkState()
{
    static uint8_t state[BUTTON_NUM_DEBOUNCE_CHECKS] = { 0 };
    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_isPushButtonPressed()) {
        state[index] |= BUTTON_DEBOUNCE_POS_PUSHBUTTON;
    }
    if (button_isRotaryButtonPressed()) {
        state[index] |= BUTTON_DEBOUNCE_POS_ROTARYBUTTON;
    }

    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 as described in the text
}
```