Exercise 4: Timers, Output Compare and Input Capture

Overview

Microcontrollers usually have at least one timer. Timers are counters that are either incremented or decremented at a fixed frequency. The timer frequency is usually configurable. When reaching zero or a user-defined value, timer interrupts can be generated. Although the number of hardware timers is limited, we can use a single hardware timer to implement an extensible set of software timers.

Timers in ATmega168

ATmega168 has three hardware timers, namely Timer0, Timer1 and Timer2. Every timer has a counter register that holds the value that changes in every timer clock cycle. Timer0 is an 8-bit general purpose timer, which means that the counter register is 8 bits wide. Hence, the counter value is between 0 and 255. Timer1 is 16-bit wide and Timer2 is another 8-bit timer, but it offers some additional functionality.

The timers can directly be driven by the system clock, which provides the highest resolution and fastest counting. Alternatively, the timer clock can be scaled down by using a so-called *prescaler*. This is basically a number that indicates how many system clock ticks should pass until the timer is incremented. In ATmega168, Timer0 and Timer1 share the same prescaler module, which allows the timers to run at 8, 64, 256 and 1024 times slower frequency, respectively. Timer2 has its own prescaler with additional factors of 32 and 128 and can even be clocked by an external (asynchronous) clock or event source.

During this session we will mostly use Timer1, since it provides the highest range of values.

Output Compare Unit

Timer1 has an output compare unit, which continuously compares the current timer counter register (`TCNT1`) with the *Output Compare Registers* (`OCR1A` and `OCR1B`). When `TCNT1` equals one of the registers, the comparator signals a match, which will set the Output Compare Flag (`OCF1x`) and generate an interrupt if enabled. Note that `OCR1A` and `OCR1B` work independently of each other and separate ISRs can be implemented for them.

Exercise 4.1

a) Which types of interrupts can be generated by Timer1? How can they be enabled / disabled? What is the name to use for the ISR for output match interrupt A and B?

b) The timer in ATmega168 can be configured to have several *Waveform Generation Modes* (WGM). In which registers are the Waveform Generation Mode bits located?

For exercises **4.1**, **4.2**, **4.3** and **4.4**, we will use the WGM 4.

c) For WGM 4, which register controls the *TOP* (maximum) value of Timer1? What will happen in case the counter reaches TOP?

d) Assume we choose the timer clock to be $\mathrm{clk}_{\mathrm{I/O}}/256$, where $\mathrm{clk}_{\mathrm{I/O}}$ is 16 MHz. Which value should we store in `OCR1A` in order to generate a sequence of timer interrupts with an interval of 1 second?

Exercise 4.2

a) Implement a `timer_init()` function that initializes the timer and a main program that generates an interrupt for a real time interval of 500 ms. Use the output compare unit for the implementation. Let an LED blink at the respective frequency.

b) In the previous sessions, we used an imprecise `wait()` function. Using Timer1, develop an accurate `timed_wait()` function that interprets the parameter as milliseconds and waits for the given time interval to pass. Modify the above program accordingly.

c) Develop an application that implements at least three software timers with different frequencies using Timer1. The respective counter of the software timers should be updated within the ISR of the hardware timer. Demonstrate the different frequencies using LEDs.

Input Capture Unit

Timer1 has an input capture unit that can store the current value of the Timer1 counter when an external event happens. This means that we can tell when exactly the event happened without having to care about how much time has passed between the event and our code handling it.

This feature also allows calculation of the frequency of external signals. The pin used to capture external events is called *Timer/Counter1 Input Capture Input* (`ICP1`). When an event is detected at `ICP1`, the current 16-bit value in the timer counter register (`TCNT1`) is copied to the *Input Capture Register* (`ICR1`). This value then represents the timestamp of the event. Note that `ICR1` will be overwritten when subsequent events happen. When enabled, an interrupt will be generated upon capturing an event.

Exercise 4.3

a) Which pin is used as `ICP1` in ATmega168?

b) Input capture is edge sensitive. How to set it to be rising edge or falling edge sensitive?

Exercise 4.4

a) Implement an application to measure the time interval between a button is pressed and released. Display the measured result (in ms) in the debug console. Note that you should adapt the prescaler for this to work properly for longer time intervals. It is acceptable if your program works for button press times up to 4 seconds.

b) Implement an application that will wait a random amount of time between 2 and 10 seconds (see hints section) and switch an LED on after the interval has passed. Then, measure the time until the user presses a button (i.e., the reaction time). Be sure to precisely measure the time by using the input capture unit. Output the reaction time in milliseconds to the console.

c) Optional: Improve your programs from exercises **4.4 a)** and **4.4 b)** to handle longer time intervals by enabling and counting timer overflow interrupts.

Hints

- It is actually not so easy to generate numbers that are somewhat random on a microcontroller. See http://www.rn-wissen.de/index.php/Zufallszahlen_mit_avr-gcc (unfortunately only in German) for starting points.

Clock Sources

Until now, we did not care much about how the microcontroller is clocked. We just assumed that a clock signal of 16 MHz is available. However, in real applications, the type and frequency of a clock source is very important, because it influences accuracy and power consumption.

But what actually is a clock signal? Figure 1 depicts three different types of clock signals. All clock signals are periodic, but depending on the method used to generate them, they may look quite different.

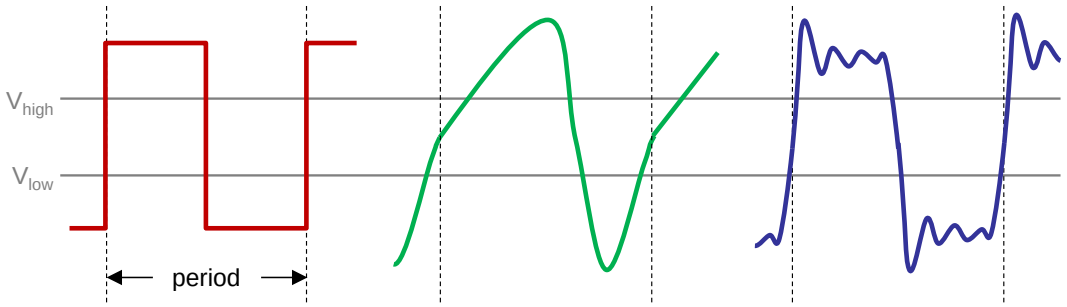

Figure 1: Various Clock Signals

The left part of Figure 1 shows a rectangular signal which can only be generated by logic circuits. The other two signals are generated by oscillators. The signal on the right is typically generated by crystal oscillators. Note that it does not matter how the clock signal is actually shaped. The only requirement is that it oscillates between two states, which are characterized by the fact that the "high" state clearly has a voltage value higher than a certain threshold V_{high} and the "low" state clearly has a voltage value lower than a certain threshold V_{low} (with $V_{\text{high}} > V_{\text{low}}$).

Exercise 4.5

a) Two of the most famous types of clock sources are RC circuits and crystal oscillators. Do some research on the Internet to find out how they are built and list their advantages and disadvantages.

b) In which of the following applications would it be meaningful to use simple RC circuits as clock sources? Explain your decision.

- **i)** A wall clock that shows the current time of the day.
- **ii)** A sensor node that measures the current temperature and transfers the value to a PC over UART every 5 minutes.
- **iii)** A controller for a model airplane.
- **iv)** A simple calculator.

Exercise 4.6 (optional)

ATmega168 can be configured to use an internal or external clock source. Until now, we have been using an external clock source of 16 MHz provided by the crystal oscillator on the *STK500* board (the silver component between the processor and the *RS232 CTRL* port). However, ATmega168 also provides an internal RC oscillator. In this application, we will try to compare the quality of these two oscillation sources.

Start by changing the following settings:

- Connect to *STK500* from *AVR Studio*.
- On the *Fuses* tab, click on *Read*.
- The `SUT_CKSEL` value should read *Ext. Crystal Osc. 8.0- MHz; Start-up time PWRDWN/RESET: 16K CK/14 CK + 65 ms* (this should be the last item in the drop-down list).
- Please be very careful now to select the correct new value, otherwise you might render the chip unusable. Change the value for `SUT_CKSEL` to *Int. RC Osc. 8 MHz; Start-up time PWRDWN/RESET: 6 CK/14 CK + 65 ms*.
- Click on *Program*.

You have now changed the microcontroller's oscillator to an 8 MHz internal RC circuit based oscillator. The value you have programmed is stored in a special memory in the chip called the *fuses*. The fuses define basic settings for the microcontroller's operation. Changes to the fuses will remain in place, that's why it is dangerous if one sets them to a wrong value. To undo your changes after this exercise, reset `SUT_CKSEL` to the value mentioned above.

For being able to estimate the quality of the internal RC oscillator, we need a reference clock. We will be using the external 16 MHz crystal oscillator for this purpose.

a) Implement a `timer2_init()` function that initializes Timer2 for use in asynchronous mode (use an external clock source on `TOSC1`) and a prescaler of 256.

b) Create an application that initializes Timer2, defines a Timer Overflow ISR and enables the respective interrupt.

c) How often will the Timer Overflow interrupt fire in this configuration at a clock of 16 MHz?

d) Design a program that measures variations in the frequency of the internal RC oscillator in relation to the external crystal oscillator. Follow these steps: Configure Timer2 to use the external clock source from the 16 MHz crystal oscillator on the STK500 board with prescaler 256 (i.e., frequency 62.5 kHz). Configure Timer1 to use the clock from the internal 8 MHz RC oscillator with prescaler 1. Count the number of Timer1 ticks between the clock overflows of Timer2. In the ideal case, we would expect the result to be 62,500, but of course there will be some overhead (e.g., for executing the ISR). Hence, we are interested in the relative change between multiple measurements rather than in the absolute value. Perform multiple measurements and store them in an array for later analysis. When all measurements have been recorded, output some statistics. Make sure your program has a deterministic runtime for each call of the ISR. Your application should output the results including the mean tick count of the "fast counter" (over all measurements) as well as the absolute difference and the relative error in percent (for each measurement) to the debug console, for example:

```
Measurements:
 # 1: 32829 (mean+5.30 = +0.01615%)
 # 2: 32819 (mean-4.70 = -0.01432%)
 :     :       :            :
 # 9: 32845 (mean+21.30 = +0.06489%)
 #10: 32831 (mean+7.30 = +0.02224%)
Mean: 32823.70
```

Note

- Exercise 4.6 of course can not give us confidence whether the internal RC oscillator or the external crystal oscillator shows these variations. We can however assume that the crystal oscillator is much more precise and that most of the error is introduced by the RC oscillator.