Lab 3: Timers

Learning objectives

After completing this lab you will be able to:

- Use `#define` compiler directives
- Use internal microcontroller timers
- Understand overflow
- Combine different interrupts

The purpose of the laboratory exercise is to understand the function of the interrupt, interrupt service routine, and the functionality of timer units. Another goal is to practice finding information in the MCU manual; specifically setting timer control registers.

Table of contents

Components list

- Arduino Uno board, USB cable
- Breadboard
- 2 LEDs or 1 two-color LED
- 2 resistors
- 1 push button
- Jumper wires

Pre-Lab preparation

Consider an *n*-bit number that we increment based on the clock signal. If we reach its maximum value and try to increase it, the value will be reset. We call this state an **overflow**. The overflow time depends on the frequency of the clock signal, the number of bits, and on the prescaler value:

$$t_{OVF} = \frac{1}{f_{CPU}} \cdot 2^{nbit} \cdot prescaler$$

> **Note:** The equation was generated by Online LaTeX Equation Editor using the following code.
>
> ```
> t_{OVF} = \frac{1}{f_{CPU}}\cdot 2^{nbit}\cdot prescaler
> ```

1. Calculate the overflow times for three Timer/Counter modules that contain ATmega328P if CPU clock frequency is 16 MHz. Complete the following table for given prescaler values. Note that, Timer/Counter2 is able to set 7 prescaler values, including 32 and 128 and other timers have only 5 prescaler values.

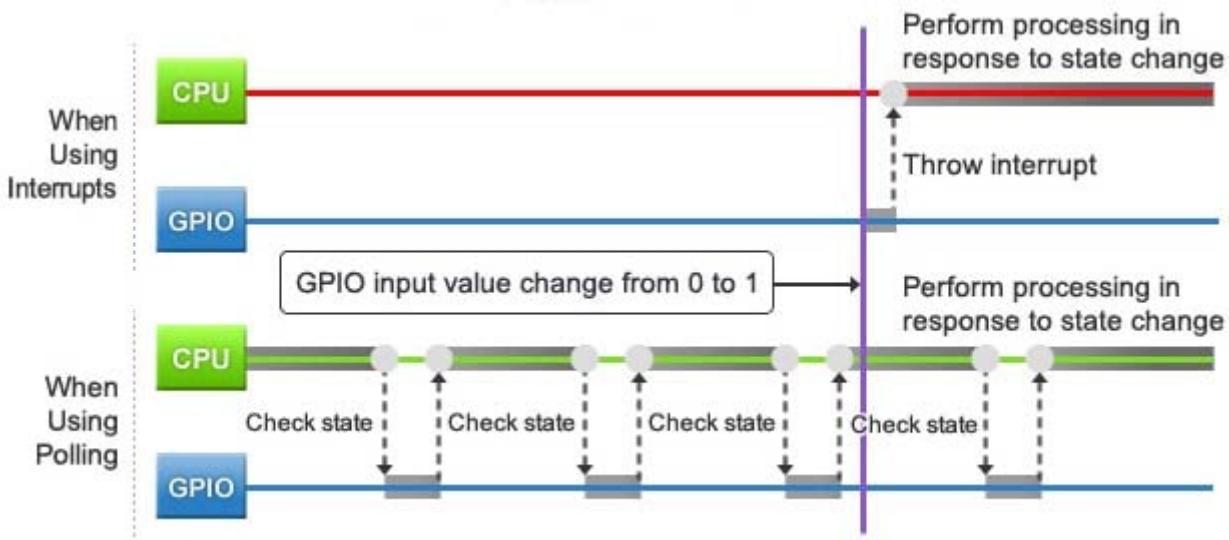
Module	Number of bits	1	8	32	64	128	256	1024
Timer/Counter0	8	16u	128u	--		--		
Timer/Counter1	16			--		--		
Timer/Counter2	8							

Part 1: Polling and interrupts

The state of continuous monitoring of any parameter is called **polling**. The microcontroller keeps checking the status of other devices; and while doing so, it does no other operation and consumes all its processing time for monitoring [3].

While polling is a straightforward method for monitoring state changes, it comes with a trade-off. If the polling interval is too long, there may be a significant delay between the occurrence and detection of a state change, potentially leading to missing the change entirely if the state reverts before the next check. On the other hand, a shorter interval provides quicker and more dependable detection, but it also consumes considerably more processing time and power, as there are more unsuccessful checks.

An alternative approach is to employ **interrupts**. In this approach, a state change triggers an interrupt signal that prompts the CPU to pause its current operation (while preserving its current state), execute the interrupt-related processing, and subsequently restore its prior state before resuming from where it had been interrupted.



An interrupt is a fundamental feature of a microcontroller. It represents a signal sent to the processor by hardware or software, signifying an event that requires immediate attention. When an interrupt is triggered, the controller finishes executing the current instruction and proceeds to execute an **Interrupt Service Routine (ISR)** or Interrupt Handler. ISR tells the processor or controller what to do when the interrupt occurs [4]. After the interrupt code is executed, the program continues exactly where it left off.

Interrupts can be set up for events such as a counter's value, a pin changing state, receiving data through serial communication, or when the Analog-to-Digital Converter has completed the conversion process.

See the ATmega328P datasheet (section **Interrupts > Interrupt Vectors in ATmega328 and ATmega328P**) for sources of interruptions that can occur on ATmega328P. Complete the selected interrupt sources in the following table. The names of the interrupt vectors in C can be found in C library manual.

Program address	Source	Vector name	Description
0x0000	RESET	--	Reset of the system
0x0002	INT0	`INT0_vect`	External interrupt request number 0
	INT1		
	PCINT0		
	PCINT1		
	PCINT2		
	WDT		
	TIMER2_OVF		
0x0018	TIMER1_COMPB	`TIMER1_COMPB_vect`	Compare match between Timer/Counter1 value and channel B compare value
0x001A	TIMER1_OVF	`TIMER1_OVF_vect`	Overflow of Timer/Counter1 value
	TIMER0_OVF		
	USART_RX		
	ADC		
	TWI		

All interrupts are disabled by default. If you want to use them, you must first enable them individually in specific control registers and then enable them centrally with the `sei()` command (Set interrupt). You can also centrally disable all interrupts with the `cli()` command (Clear interrupt).

Part 2: Synchronize repositories and create a new project

1. In your working directory, use **Source Control (Ctrl+Shift+G)** in Visual Studio Code or Git Bash (on Windows) or Terminal (on Linux) to update the local repository.

   > **Help:** Useful bash and git commands are `cd` - Change working directory. `mkdir` - Create directory. `ls` - List information about files in the current directory. `pwd` - Print the name of the current working directory. `git status` - Get state of working directory and staging area. `git pull` - Update local repository and working folder.

2. In Visual Studio Code create a new PlatformIO project `lab3-timers` for `Arduino Uno` board and change project location to your local repository folder `Documents/digital-electronics-2`.

3. IMPORTANT: Rename `LAB3-TIMERS > src > main.cpp` file to `main.c`, ie change the extension to `.c`.

Part 3: Timer overflow

A timer (or counter) is an integral hardware component in a microcontroller unit (MCU) designed for measuring time-based events. The ATmega328P MCU features three timers, designated as Timer/Counter0, Timer/Counter1, and Timer/Counter2. Timer0 and Timer2 are 8-bit timers, whereas Timer1 is a 16-bit timer.

The counter increments in alignment with the microcontroller clock, ranging from 0 to 255 for an 8-bit counter or 65,535 for a 16-bit counter. If counting continues, the timer value overflows to the default value of zero. Various clock sources can be designated for each timer by utilizing a CPU frequency divider equipped with predetermined prescaler values, including 8, 64, 256, 1024, and other options.

1. The timer modules can be configured with several special purpose registers. According to the ATmega328P datasheet (eg in the **8-bit Timer/Counter0 with PWM > Register Description** section), which I/O registers and which bits configure the timer operations?

Module	Operation	I/O register(s)	Bit(s)
Timer/Counter0	Prescaler 8-bit data value Overflow interrupt enable		
Timer/Counter1	Prescaler 16-bit data value Overflow interrupt enable	TCCR1B TCNT1H, TCNT1L TIMSK1	CS12, CS11, CS10 (000: stopped, 001: 1, 010: 8, 011: 64, 100: 256, 101: 1024) TCNT1[15:0] TOIE1 (1: enable, 0: disable)
Timer/Counter2	Prescaler 8-bit data value Overflow interrupt enable		

2. Copy/paste template code to `LAB3-TIMERS > src > main.c` source file.

3. In PlatformIO project, create a new folder `LAB3-TIMERS > lib > gpio`. Copy your GPIO library files `gpio.c` and `gpio.h` from the previous lab to this folder.

4. In PlatformIO project, create a new file `LAB3-TIMERS > include > timer.h`. Copy/paste header file to `timer.h`. See the final project structure:

```
LAB3-TIMERS          // PlatfomIO project
├── include          // Included file(s)
│   └── timer.h
├── lib              // Libraries
│   └── gpio         // Your GPIO library
│       ├── gpio.c
│       └── gpio.h
├── src              // Source file(s)
│   └── main.c
├── test             // No need this
└── platformio.ini   // Project Configuration File
```

To simplify the configuration of control registers, we defined Timer/Counter1 macros with meaningful names in the `timer.h` file. Because we only define macros and not function bodies, the `timer.c` source file is **not needed** this time!

5. Go through the files and make sure you understand each line. Build and upload the code to Arduino Uno board. Note that `src > main.c` file contains the following:

```
#include <avr/interrupt.h>  // Interrupts standard C library for AVR-
GCC
#include <gpio.h>           // GPIO library for AVR-GCC
#include "timer.h"          // Timer library for AVR-GCC

int main(void)
{
    ...
    // Enable overflow interrupt
    TIM1_OVF_ENABLE
    ...
    // Enables interrupts by setting the global interrupt mask
    sei();
    ...
}

// Interrupt service routines
ISR(TIMER1_OVF_vect)
{
    ...
}
```

6. In `timer.h` header file, define similar macros also for Timer/Counter0 and Timer/Counter2. On a breadboard, connect a two-color LED (3-pin LED) or two LEDs and resistors to pins PB2 and PB3. Modify `main.c` file, and use three interrupts for controlling all three LEDs (one on-board and two off-board). Build and upload the code into ATmega328P and verify its functionality.

7. (Optional) Consider an active-low push button with internal pull-up resistor on the PD2 pin. Use Timer0 4-ms overflow to read button status. If the push button is pressed, turn on `LED_RED`; turn the

LED off after releasing the button. Note: Within the Timer0 interrupt service routine, use a read function from your GPIO library to get the button status.

Part 4: Extend the overflow

1. Use Timer/Counter0 16-ms overflow and toggle `LED_RED` value approximately every 100 ms (6 overflows x 16 ms = 100 ms).

   FYI: Use static variables declared in functions that use them for even better isolation or use volatile for all variables used in both Interrupt routines and main code loop. According to [7] the declaration line `static uint8_t no_of_overflows = 0;` is only executed the first time, but the variable value is updated/stored each time the ISR is called.

```
ISR(TIMER0_OVF_vect)
{
    static uint8_t no_of_overflows = 0;

    no_of_overflows++;
    if (no_of_overflows >= 6)
    {
        // Do this every 6 x 16 ms = 100 ms
        no_of_overflows = 0;
        ...
    }
    // Else do nothing and exit the ISR
}
```

2. Reduce the overflow time by storing a non-zero value in the Timer/Counter0 data register TCNT0 after each overflow.

```
ISR(TIMER0_OVF_vect)
{
    static uint8_t no_of_overflows = 0;

    no_of_overflows++;
    if (no_of_overflows >= 6)
    {
        no_of_overflows = 0;
        ...
    }
    // Change 8-bit timer value anytime it overflows
    TCNT0 = 128;
    // Overflow time: t_ovf = 1/f_cpu * (2^bit-init) * prescaler
    // Normal counting:
    // TCNT0 = 0, 1, 2, ...., 128, 129, ...., 254, 255, 0, 1
    //         |--------------------------------------|
    //                          16 ms
    // t_ovf = 1/16e6 * 256 * 1024 = 16 ms
    //
```

```
    // Shortened counting:
    // TCNT0 = 0, 128, 129, ...., 254, 255, 0, 128, ....
    //        |--------------------------|
    //                    8 ms
    // t_ovf = 1/16e6 * (256-128) * 1024 = 8 ms
}
```

3. After completing your work, ensure that you synchronize the contents of your working folder with both the local and remote repository versions. This practice guarantees that none of your changes are lost. You can achieve this by using **Source Control (Ctrl+Shift+G)** in Visual Studio Code or by utilizing Git commands.

> **Help:** Useful git commands are `git status` - Get state of working directory and staging area. `git add` - Add new and modified files to the staging area. `git commit` - Record changes to the local repository. `git push` - Push changes to remote repository. `git pull` - Update local repository and working folder. Note that, a brief description of useful git commands can be found here and detailed description of all commands is here.

(Optional) Experiments on your own

1. In `timer.h` header file, complete macros for all three timers.
2. Enhance the current application to control four LEDs in the Knight Rider style. Avoid using the delay library and instead, implement this functionality using a single Timer/Counter.
3. Use the ATmega328P datasheet (section **8-bit Timer/Counter0 with PWM > Modes of Operation**) to find the main differences between:
    - Normal mode,
    - Clear Timer on Compare mode,
    - Fast PWM mode, and
    - Phase Correct PWM Mode.
4. Finish all experiments, upload them to your GitHub repository, and submit the project link via BUT e-learning. The deadline for submitting the assignment is the day prior to the next lab session, which is one week from now.

References

1. Tomas Fryza. Schematic of Arduino Uno board
2. Microchip Technology Inc. ATmega328P datasheet
3. Renesas Electronics Corporation. Essentials of Microcontroller Use Learning about Peripherals: Interrupts
4. Tutorials Point. Embedded Systems - Interrupts
5. C library manual
6. norwega. Knight Rider style chaser

7. StackOverflow. Static variables inside interrupts
8. Tutorials Point. Arduino - Pulse Width Modulation
9. Tomas Fryza. Useful Git commands
10. Goxygen commands