Lab 5: Timers and Interrupts

Introduction

The objective of this lab is to investigate how we can measure and use *time* in the ATmega328P/PB microcontroller. A lot of embedded systems require accurate timing in their operation, and most microcontrollers feature multiple timer peripherals for use in this domain. Timer peripherals can be used to produce required delays, as well as monitor or trigger time-based events. As peripherals, the operation of each timer can be thought as independent of program execution once they are configured/invoked.

For the purposes of this lab, we will be using Timer0 with polling and interrupts. We will also warn here that sometimes datasheets contain errors or are missing information. This is especially true for new devices. Make sure to use the latest version of the datasheet from the Microchip website.

Read the entire lab manual before you start. This lab should take you approximately **4 hours**. There is a pre-lab, four compulsory parts and an optional activity. Before starting the lab, pull the lab repository from GitHub as it contains the Proteus model(s). Remember to commit and push your saved work (code and Proteus simulations) to the lab repository on GitHub after completing each task. **The final answers should be written in this document, with supporting calculations shown in your logbook.**

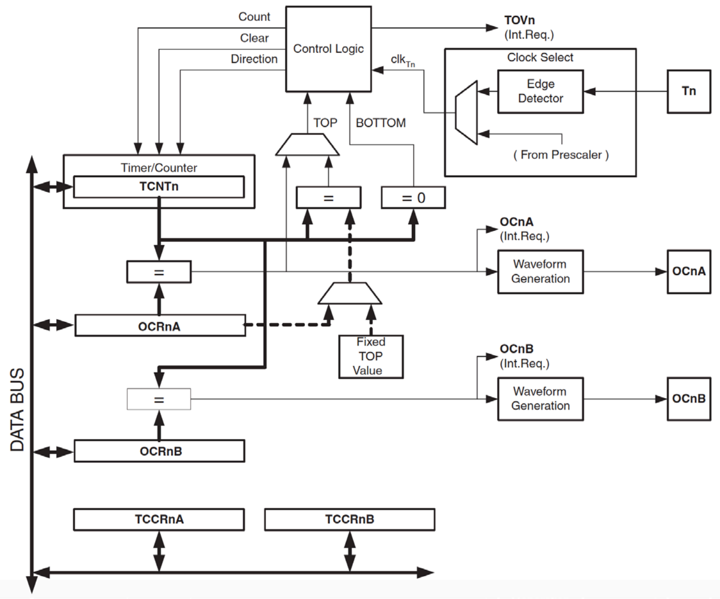
Pre-Lab: Busy-Waiting to Flash an LED

At this point, you should be familiar with the ATmega328P/PB, Atmel/Microchip Studio, Xplained Mini, Emulator Board, and Proteus. Before the lab, we will refresh our understanding of external input/output (I/O), as well as some elements of timing (Note: This part is similar, but not identical, to the pre-lab for Lab 4):

1) Open the Lab 5 Proteus project (Lab5_Proteus_All.pdsprj).
    a. Note the differences between the Lab 4 and Lab 5 projects.
2) Create a new GCC C Executable Project in Atmel/Microchip Studio, selecting ATmega328P as the device family.
3) Use the appropriate DDRn registers to set PORTB to be outputs, and PORTC and PORTD to be inputs.
4) Using <util/delay.h> and _delay_ms(time), toggle the LED in the Proteus project so that it flashes with a **2Hz frequency and 75% duty cycle** continuously (i.e. in the while loop turn-on the LED for 0.375s and turn-off the LED for 0.125s). Use the Proteus schematic to determine which pin you should use to control the LED. You may want to refer to the pre-lab of Lab 4 for some help on how to generate the delays.
5) Build, download, and run the program. Visually inspect that the LED seems to be flashing at a reasonable rate (note that this will depend on how capable your simulating PC is). Use Proteus' integrated oscilloscope to probe the LED pin and confirm that the time period is correct.

Part 1: What is a Timer Peripheral?

A timer peripheral is a component inside a microcontroller responsible for counting clock cycles (or external input pulses), up to, or down from, a given number. In an ATmega328P/PB, there are several timers. Timer0 (also called TC0) is an 8-bit timer. There is also Timer1 which is 16-bit and Timer2 which is 8-bit. The ATmega328PB has 2 additional 16-bit timers, Timer3 and Timer4. The ATmega328P/PB datasheet provides the following system diagram for Timer0:



As can be seen, the counter exists primarily as an 8-bit wide register, TCNTn, with control logic for dictating when and how that counter should count, and comparison logic to decide what action to take when TCNTn reaches a specific value. Note that the *"n"* in a given register name represents the timer number, so for Timer0, the register names would be TCNT0, OCR0A, etc.
The timer (configuration) is managed via a set of control registers, TCCRnA and TCCRnB, and comparisons (match values) are managed via the control registers OCRnA and OCRnB.

Q 1.1: Based on the diagram above and the information provided in the ATmega328P datasheet, are the two OCRnX registers (i.e. ORCnA and OCRnB) needed for all Timer0 configurations?

No, When run in Normal Mode

Q 1.2: Based on the lecture slides, ATmega328P datasheet, COMPSYS201 notes, and other online resources, find and then state in your own words the difference between the ***range*** *and* ***resolution*** *of a given timer:*

Resolution is the one timer clock cycle

Range is the max time interval timer can measure.

Q 1.3: What is the purpose of the prescaler used with the clock source (also shown in the block diagram)?

Allowing timer to measure longer period

Q 1.4: If the clock rate of the ATmega328P is 2MHz, and the clock selection bits in TCCR0B are set to "100",

a) *What is the prescaler set to?* 256
b) *What is the resolution of Timer0?* 128 u *microseconds*
c) *What is the maximum range of Timer0?* 32640 *microseconds*

One of the simplest methods for timing using a timer is CTC mode, which stands for "Clear Timer on Compare match". In the context of Timer0, CTC mode compares the timer count register with the value in the compare register OCR0A. If they equal, the timer is reset, and the "output compare A match" flag is set.

Q 1.5: We want Timer0 to use CTC mode to set the "output compare A match" flag every ~10 milliseconds (9.984ms to be exact since we can only load an integer number to OCR0A). What should the following registers be set to (some bits have already been filled in for you)?

a) TCCR0A

COM0A1	COM0A0	COM0B1	COM0B0			WGM1	WGM0
0	0	0	0	0	0	1	0

b) TCCR0B

FOC0A	FOC0B			WGM02	CS02	CS01	CS00
0	0	0	0	0	1	0	0

c) OCR0A

OCR0A7	OCR0A6	OCR0A5	OCR0A4	OCR0A3	OCR0A2	OCR0A1	OCR0A0
0	1	0	0	1	1	0	1

Timer0 resets and sets the "output compare A match" flag when TCNT0 reaches OCR0A. When this happens, an interrupt can be generated.

Q 1.6: In which bit of which register is the "output compare match A" flag?

OCF0A in TIFR0

Part 2: Controlling Timer0

Q 2.1: Using your answers thus far, create an Atmel/Microchip Studio Project, and complete the code provided below to toggle the LED that is connected to the ATmega328P in the proteus project "Lab5_Proteus_All.pdsprj". It should toggle *every ~10ms (9.984ms to be exact), and you should be using Timer0. Use polling to check the compare match flag. Skeleton code provided below will help you do this.*

This is a good place to remind you about *modularity* – ideally, we should be putting all of the functions related to the use of Timer0 in its own .c file, with a corresponding .h file. Let's say your main code is in a file called **main.c**. Create a c source file called **timer0.c**, and a corresponding header file called **timer0.h**. In the header file, you should provide function declarations (prototypes) for all functions defined in the source file. Then, we should make sure that the **timer0.h** header file is #include-d in **both** the **main.c** source file and the **timer0.c** source file. Similarly, we should also have a separate c source file called **led.c**, and a corresponding header file called **led.h**, that interacts with the LED. You may recall we also discussed this in the previous lab and gave some guidelines about what to include in a ".h" vs a ".c".

main.c

```
#define F_CPU 2000000UL

#include "timer0.h"
#include "led.h"

#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>

int main(void){
	//TODO: set direction of LED port to OUTPUT

	timer0_init();

	while(1){
		if(timer0_check_clear_compare()) {
			led_toggle();
		}
	}
}
```

timer0.h

```
#ifndef TIMER0_H_
#define TIMER0_H_

#include <stdbool.h>

//Initialize timer0 as per Part 1
void timer0_init();

//Using polling check if timer0 has reached comparison value
//if so, it will clear the compare flag and return 1
//otherwise, it returns 0
uint8_t timer0_check_clear_compare();

#endif
```

timer0.c

```
#include "timer0.h"
#include "led.h"

#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdint.h>

void timer0_init(){
	//TODO: initialise and configure timer0 to count to 10ms
}

uint8_t timer0_check_clear_compare(){
	if( ?????? & (1 << ?????? )){ //TODO: check compare flag
		//TODO: clear compare flag.
		//Note: in datasheet this is done by writing 1 to the compare flag

		return 1;
	}
	return 0;
}
```

led.h

```
#ifndef LED_H_
#define LED_H_

//Toggles the LED
void led_toggle();

#endif
```

led.c

```
#include "led.h"

#include <avr/io.h>

void led_toggle(){
	//TODO: LED toggle code
}
```

When you compile the program, the *linker* will automatically find everything and create the right references so that the functions can be used in the appropriate source files. Throughout your project, there are different microcontroller modules that are being used (e.g. USART, ADC, timers), and it is a good idea to separate the functions that are developed for each module.

Q 2.2: Compile the Atmel/Microchip Studio project, link the elf file with the ATmega328P processor in the Lab5_Proteus_All.pdsprj Proteus project, and simulate your program in Proteus. Observe that the LED is toggling every ~10ms using the integrated Proteus oscilloscope.

Part 3: Timer Interrupts

Polling is not the best use of the microcontroller's limited resources. What designers usually do instead is get the microcontroller peripheral to start the counter and count in the background while the microcontroller attends to other operations (i.e., the counting and checking work is done in parallel). When the count has reached a specific value (e.g. value stored in OCR0A in CTC mode), the peripheral will *interrupt* normal operation and transfer control to a special *interrupt service routine* (ISR) function where you may perform specific tasks.

Note that an ISR function should be kept as short as possible. For example, you should not try to *print* anything to the terminal from an ISR, as this takes a relatively "long" time. One way to debug an ISR is by toggling a pin to indicate if it is being executed. Proteus also provides a number of methods to debug an ISR.

Q 3.1: Change the embedded program you developed previously to toggle the LED using an Output Compare A match interrupt. To do this, firstly, enable the timer0 interrupt in your timer0_init(), add the ISR function to your timer0.c, and in here toggle the LED:

timer0.c (added ISR function, enable timer0 interrupt in init)

```
ISR(TIMER0_COMPA_vect) {    //This ISR function is called when timer0 reaches
                            //compare value, compare flag is automatically cleared
       led_toggle();
}

void timer0_init() {
       //TODO: initialise and configure timer0 to interrupt every 10ms
}
```

Now, enable the global interrupt configuration from your main():

main.c (main function only)

```
int main(void)
{
       //TODO: set direction of LED port to OUTPUT

       timer0_init();
       sei();         //This special function call enables the use of interrupts

       while (1) { }
}
```

Q 3.2: Compile the Atmel/Microchip Studio project, link the elf file with the ATmega328P processor in the Lab5_Proteus_All.pdsprj Proteus project, and simulate your program in Proteus. Observe that the timer0 toggles the LED appropriately using the integrated Proteus oscilloscope.

Recall the range of Timer0 (that you deduced in Q1.4). This is quite a short period of time! Sometimes, we want a timer to generate (or measure) times longer than their range would allow. We can solve this problem by adding a counter variable in the ISR to keep track of how many times it has been triggered, and only perform some action when that counter reaches a specific value.

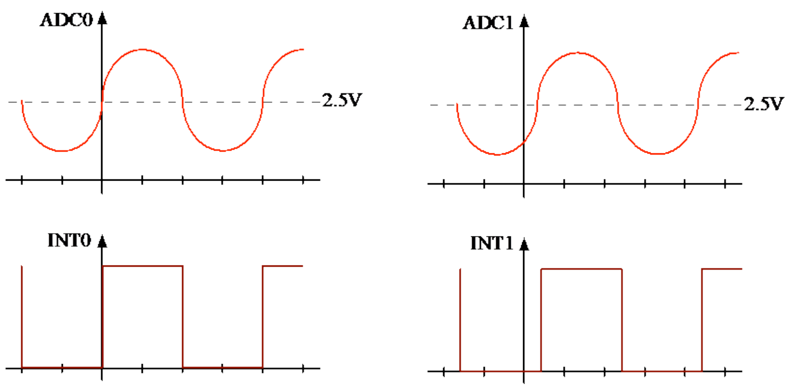
Q 3.3: In previous steps, you have set the value of OCR0A so that the ISR is called every ~10 ms. Add a software counter (variable) in the ISR, and use that to change the behaviour of your program such that the LED is instead toggled every ~100ms.

There are other methods of performing this task. For instance, a larger prescaler may be used. Alternatively, other (larger) timers may be used, such as Timer1, which has a 16-bit wide counter instead of an 8-bit wide counter.

Q 3.4: What is the maximum range of Timer1? Assume the system clock frequency is 2MHz. 33.55 s

Part 4: Measuring Time

So far we have used a timer to implement a periodic task, where we toggled a LED at a fixed frequency. This functionality will be useful in your project as you will have to write data to the display at regular intervals. More on this will be discussed in the next lab.
We can also use timers to measure a time interval. Let's look at the two signals you investigated in Lab 4 Part 5. These two signals, which were connected to ADC0 and ADC1, represent the voltage across the AC source and the load current. As explained in Lab 4, in the real design, these two signals will be coming from your signal conditioning circuitry that processes the signals derived from the voltage divider and the current sense resistor. In the Proteus simulation however, to keep it simple, we emulated these signals using sine wave sources, where you can adjust the amplitude and phase as per your analogue design (for example based on numbers you get from LTspice).
As discussed in the lectures, in some cases it may be useful to measure the power factor angle. We can do this by measuring the time between a zero-crossing of the signal representing the voltage (i.e. signal at ADC0) and the zero-crossing of the signal representing the current (i.e. signal at ADC1). In addition, the zero crossings can be used to determine the frequency of the voltage/current signal. In Lab5_Proteus_All.pdsprj, the outputs of the voltage and current zero-crossing detectors are connected to INT0 (PD2) and INT1 (PD3), respectively. Though in practice these zero crossing are detected using comparators. In Proteus, to again keep things simple, we are using ideal zero-crossing detectors on each signal. As you would expect, the outputs of these two zero-crossing detectors are two square-waves and the rising/falling edges of each square-wave aligns with the zero-crossings of the corresponding signal. The signals at ADC0, ADC1, INT0 and INT1 are shown below for clarity:



Q 4.1: Modify your code so that the timer is started on the rising edge of INT0, and stopped on the falling edge of INT0. You can poll the input at INT0, or you can use interrupts. Calculate the frequency of the waveform and send the result over the UART to be displayed on a terminal window. Compile the Atmel/Microchip Studio project and simulate your program in Proteus. What frequency is the waveform on INT0? 504.03 Hz

Q 4.2: What is the lowest frequency waveform the timer could measure? Why? Assume prescaler is 256 and you are measuring the full period (i.e., time between two rising edges).

30.52 Hz, Timer1 has only 8 bies so 1 tick eaks 128us with prescaler=256. The calculation shows in log book.

Q 4.3: Load your code on to the Xplained Mini. If you have a Xplained Mini 328PB, remember to change the processor to an ATmega328PB and recompile your code. Make sure the Xplained Mini is plugged on to the Emulator Board. On PuTTY or other terminal program observe the messages printed. What frequency is the waveform on INT0? 504.03 Hz

Optional Task: Auto-Triggering the ADC

In Lab 4 we operated the ADC in the single conversion mode taking a sample every ~104µs (i.e., ~13/125kHz). However, this is not the best way to do it. In a real design, often, ADC conversions are triggered by a timer. For this reason, even a basic microcontroller, like the ATmega328P/PB, provides a means to configure the ADC to automatically trigger the start of a conversion based on a timer event like a compare match or an overflow. We can use this functionality to take a reading of ADC0 and ADC1 every 100µs, very accurately. Note that your project specifications also state ADC results should be taken at 100µs intervals.

Modify your code from Lab 4 using some of the modules you developed in this lab to configure the ADC to auto-trigger every 100µs based on a Timer0 Output Compare A match event. In the *ADC Conversion Complete* ISR, read the values of ADC0 and ADC1, alternating between these two channels after each read. As you did in Lab 4, your program should read 40 samples from each ADC channel and store these values in two arrays (one for each ADC channel). Once you have read all the 80 samples, you should transmit this data over UART to be displayed on a terminal program. You should also aim to calculate the power represented by these two signals using the C algorithm you developed to calculate the power as part of the Lab 4 Optional Task. Finally, copy and paste the voltage and current data to Excel and verify the accuracy of your power calculation.

<u>To get signed off for the lab:</u>

- Record all the workings in hardcopy or in digital format so we can give you marks for workings if final answers are not correct
- Modify the simulation models provided as per the lab tasks and save
- Commit and push your saved work to the lab repository on GitHub after completing each task and make sure it is up to date
- Complete this document summarising your final answers, commit and push to lab repository on GitHub
- Update logbook indicating yours and teams progress, meeting notes, etc.
- Go to your assigned interview session to check-in with a TA who will check your solutions and ask a few questions
- Report on your weekly progress