ECED3204 – Lab #5

STUDENT NAME(s): .

STUDENT NUMBER(s): B00 .

# Pre-Lab Information

It is recommended that you read this entire lab ahead of time. Doing so will save you considerable time during the lab, as you will be required to write some simple C code during this lab!

# Overall Objective

This lab is designed to teach you about using serial communications. You will learn about using the USART module in the AVR, and using the ‘printf()’ statements from your code for debug and data transfer.

# Part #1: Timing Events

## Objective

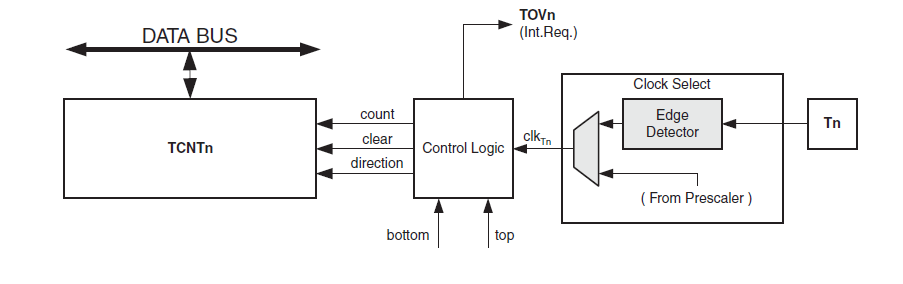
* Familiarize yourself with using a timer and interrupt.

## Required Materials

* Microprocessor Module with Programmer
* Breadboard
* USB Cable
* Power Supply
* Computer with Atmel Studio 6.2 and Programmer Utility installed
* Push Button

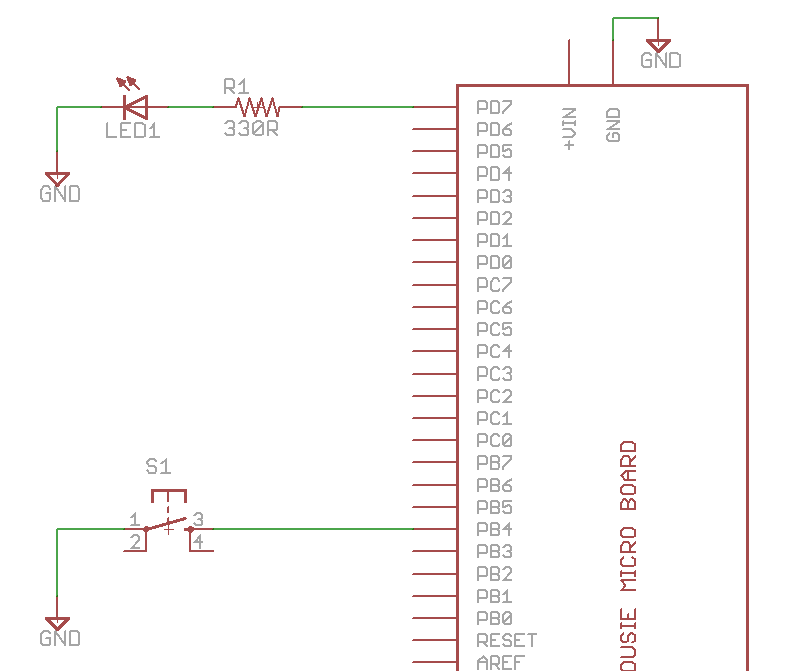
## Background

The timer system in the ATMega644P can be used for a variety of purposes. In this first part of the lab we will use the ‘overflow interrupt’ of the timer. The timer system has an 8-bit or 16-bit register (depending on the timer being used), which is incremented on a rate you can choose. See Chapter 11 of the course textbook for more information.

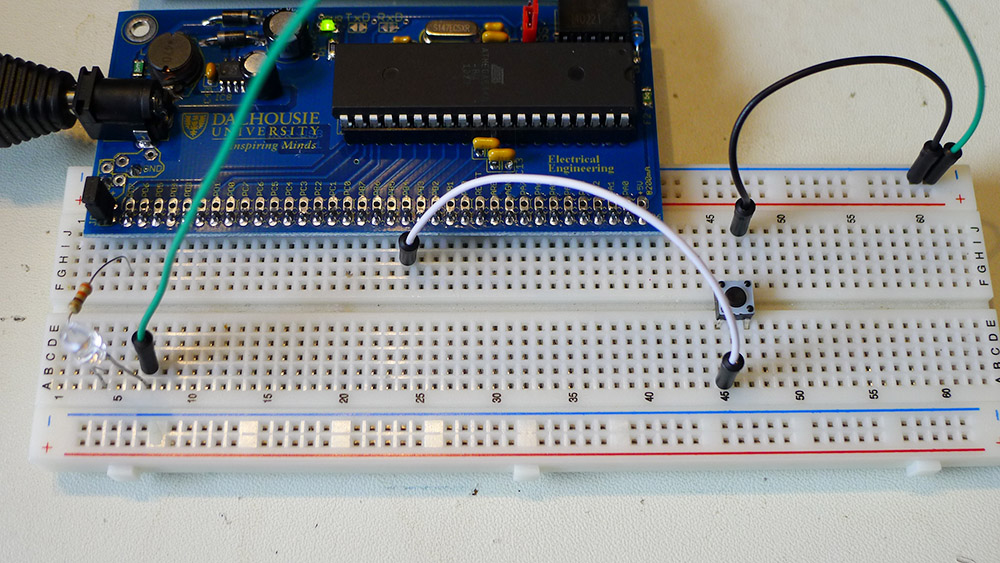


## Procedure

1. Build the same circuit from **Part #1** of **Lab #3**. For reference the schematic is shown below – we will only use the LED in Part 1, but will use the switch in Part 2.



Which might look as follows:



1. Start a new C/C++ project (see Lab #1 for details), copy the following template into it:

#include <avr/io.h>

#include <avr/interrupt.h>

volatile unsigned int tick;

#define LED\_ON() PORTD |= 1<<7

#define LED\_OFF() PORTD &= ~(1<<7)

int main(void)

{

DDRD |= 1<<7;

//Timer configuration

TCCR0A = ?

TCCR0B = ?

//Interrupt mask enables

TIMSK0 = ?

//Enable global interrupts

sei();

while(1);

}

ISR(SPECIFY\_NAME\_HERE)

{

//Code here

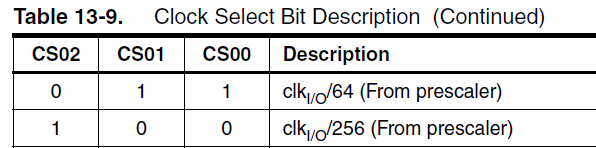
;

}

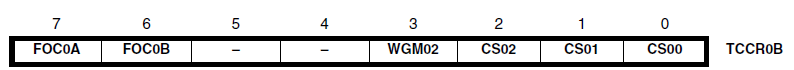
1. Before being able to compile this code, we need to finish a few things. The first is to determine the setup of TCCR0A and TCCR0B. We want the timer to overflow somewhere between 100-5000 times per second.

The clock source for the AVR is a 14745600Hz crystal (we learned this in Lab #1). If you used this to directly drive an 8-bit counter, the counter would overflow at the rate of 14745600 ÷ 28 = 14745600 ÷ 256 = 57600 times per second. That is too fast!

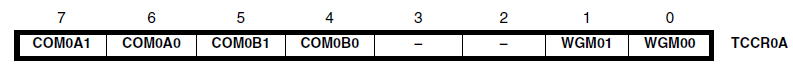
Finding Table 13-9, you can see some prescaler settings:



Using a prescaler of 64 would mean the timer now overflows 900 times per second. This tells us the setting for two bits in the TCCR0B register, as these CS00/CS01/CS02 bits are present in this register:



We will use the default (all-0) settings for TCCR0A:



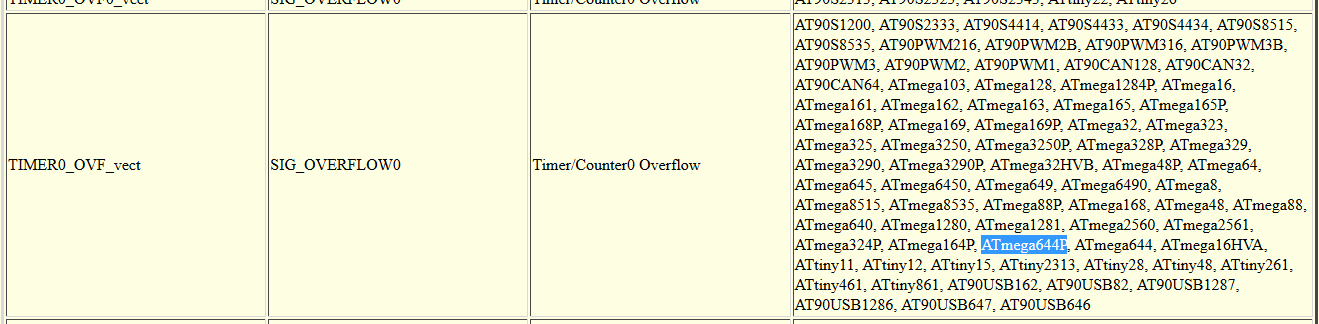
See the datasheet for a description of those bits. This means our setup area of the code looks like this:

//Timer configuration

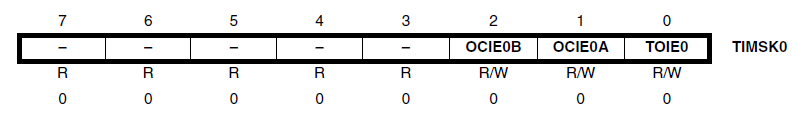
TCCR0A = 0;

TCCR0B = (1 << CS00) | (1 << CS01);

1. Next you need to fill in the SPECIFY\_NAME\_HERE section of the code, which is the interrupt vector. This routine will get called every time the interrupt occurs. You can find a list of all the names at <http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html> - you must be **VERY CAREFUL** to ensure you use a name that exists on the ATmega644P device! In this case look for the TIMER0\_OVF\_vect name, as shown in the following:



1. Finally, you need to enable the ‘overflow’ interrupt, which is done by setting bit 1 of the TIMSK0 register:



1. At this point, your code should look similar to this:

int main(void)

{

DDRD |= 1<<7;

//Timer configuration

TCCR0A = 0;

TCCR0B = (1 << CS00) | (1 << CS01);

//Interrupt mask enables

TIMSK0 |= 1 << TOIE0;

//Enable global interrupts

sei();

while(1);

}

ISR(TIMER0\_OVF\_vect)

{

//Code here

;

}

1. Finally, program the interrupt service routine (ISR) to toggle the LED. An example implementation is as follows, using the ‘tick’ variable to keep track of how many runs through the interrupt routine have occurred:

ISR(TIMER0\_OVF\_vect)

{

tick++;

if(tick == 1){

LED\_ON();

} else if (tick == 400){

LED\_OFF();

} else if (tick > 900){

tick = 0;

}

}

1. Adjust the delay to blink 5 seconds on, 5 seconds off.

# Part #2: PWM Output

## Objective

* Generate a PWM signal with a variable duty cycle.

## Required Materials

* Setup from Part #1

## Background

The timer system in the ATMega644P can be used for a variety of purposes. In this first part of the lab we will use the ‘overflow interrupt’ of the timer. The timer system has an 8-bit or 16-bit register (depending on the timer being used), which is incremented on a rate you can choose. See Chapter 11 of the course textbook for more information.

## Procedure

1. Build the same circuit from **Part #1** of this lab, or if you just completed Part #1 leave your circuit as-is.
2. Using Timer 2, generate a Pulse Width Modulation (PWM) signal of around 2-10 kHz. Output this on in PD7 (which is the OC2A pin). You will need the following references in the datasheet:
   * Section 15.11.1 – TCCR2A Register
   * Section 15.11.2 – TCCR2B Register

You should configure timer/counter 2 as the following:

* + PWM, Phase Correct (WGM2 = 0, WGM1 = 1, WGM0 = 0)
  + Clock divider = IOClock / 8
  + OC2A pin operating in non-inverting mode

The following shows the basic operating instructions:

#include <avr/io.h>

int main(void)

{

DDRD |= 1<<7;

//Set clock divider to be /8

TCCR2B = ??

//Set waveform generation mode

TCCR2A |= ??

//Set output on OC2A pin

TCCR2A |= ??

//Set PWM to half-way (50% duty cycle)

OCR2A = 127;

while(1);

}

1. The LED should be partially illuminated now. You can also check with an oscilloscope you are getting an appropriate signal.
2. Set the OCR2A register to various values in the range 0-255, and observe the effect on the output signal. Use an oscilloscope to measure how the duty cycle changes, and measure the frequency as well of the PWM output.
3. Add some code to slowly increase the value, and see what happens. For example here is a complete code listing:

#include <avr/io.h>

#include <util/delay.h>

int main(void)

{

DDRD |= 1<<7;

//Set clock divider to be /8

TCCR2B = 1<<CS21;

//Set waveform generation mode

TCCR2A |= 1<<WGM20;

//Set output on OC2A pin

TCCR2A |= 1<<COM2A1;

OCR2A = 0;

while(1){

OCR2A++;

\_delay\_ms(50);

}

}

# Part #3: Input Capture

## Objective

* Measure the time a button is pressed using the input capture

## Required Materials

* Setup from Part #1

## Background

The timer system in the ATMega644P can be used for a variety of purposes. In this first part of the lab we will use the ‘overflow interrupt’ of the timer. The timer system has an 8-bit or 16-bit register (depending on the timer being used), which is incremented on a rate you can choose. See Chapter 11 of the course textbook for more information.

# 

# Part #2: Multiplexing LEDs

## Objective

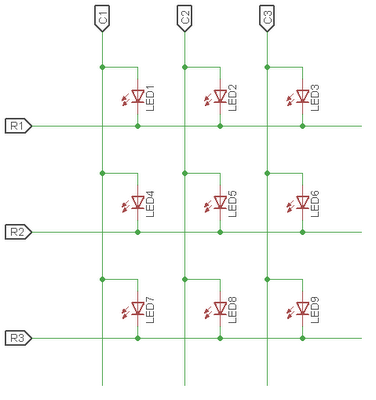
* Familiarize yourself with the idea of multiplexing LED outputs

## Required Materials

* Lab setup from Part 1 (remove push-button)
* 4x LEDs
* 2x 330 ohm resistors

## Background

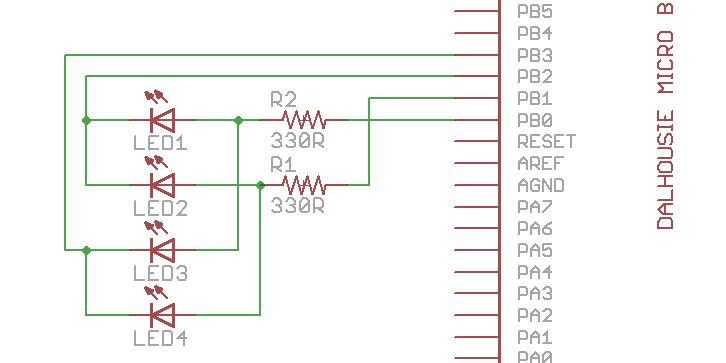
Multiplexing allows us to write or read to a larger number of components then you might otherwise be able to. This example will use a ‘multiplexing’ capability for only four LEDs. But consider the following, which shows a 3x3 array of LEDs, driven with only 6 I/O pins (instead of 9 required to drive each LED separately):



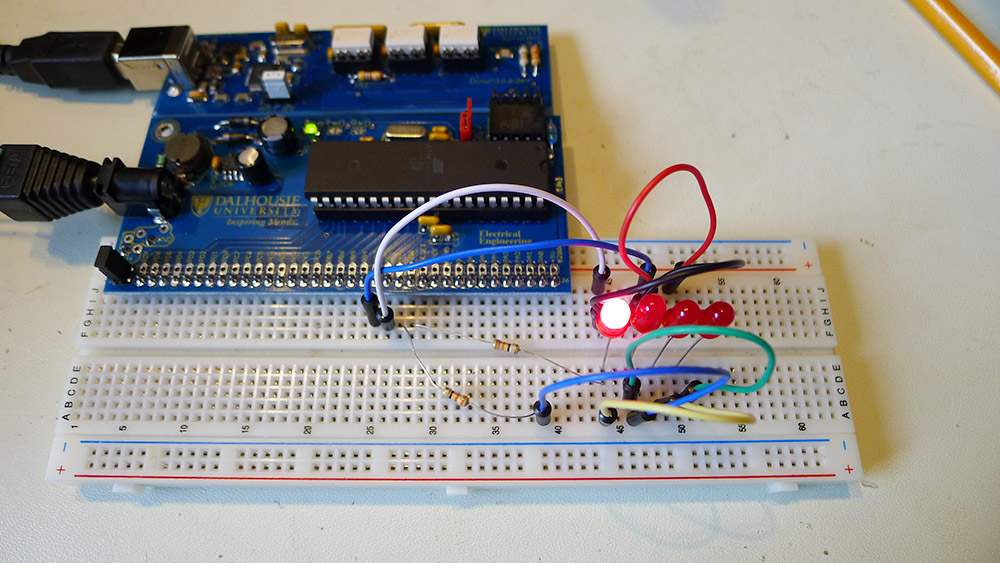
We will instead just be driving four LEDs. We don’t actually realize any pin savings with just four LEDs, but you can expand this on your own time (what fun!).

## Procedure

1. Plug the microprocessor board into the breadboard and apply power, see Lab #1 for details.
2. Wire up the following circuit on the breadboard:



Which might look something like this:



1. At this point, we need to determine how to turn on each LED. Using the schematic diagram, fill out the following table, which has a few entries given to you. The idea is to illuminate LED 0 for example, we need to set PORTB.0 high, and PORTB.2 low. That will drive current through LED0.

|  |  |  |
| --- | --- | --- |
| LED Number | PORTB Pin High | PORTB Pin Low |
| 0 | 0 | 2 |
| 1 |  |  |
| 2 | 0 |  |
| 3 |  | 3 |

1. Code the following, where the appropriate PORTx pin has been set high or low based on the above table, and will cycle through each LED:

#include <avr/io.h>

#include <util/delay.h>

void set\_led(unsigned char lednum)

{

switch(lednum){

case 0:

DDRB = 1<<0 | 1<<2;

PORTB = 1<<0;

break;

case 1:

DDRB = 1<<1 | 1<<2;

PORTB = 1<<1;

break;

case 2:

DDRB = 1<<0 | 1<<3;

PORTB = 1<<0;

break;

case 3:

DDRB = 1<<1 | 1<<3;

PORTB = 1<<1;

break;

}

}

int main(void)

{

while(1){

set\_led(0);

\_delay\_ms(2000);

set\_led(1);

\_delay\_ms(2000);

set\_led(2);

\_delay\_ms(2000);

set\_led(3);

\_delay\_ms(2000);

}

}

1. Download the above code, and confirm that your LEDs illuminate in sequence.
2. How would you illuminate multiple LEDs? The circuit itself stops you from independently illuminating LEDs, but consider what happens if you switch between two LEDs very quickly. Change the main loop to the following:

int main(void)

{

while(1){

set\_led(0);

set\_led(2);

set\_led(3);

}

}

You can play with adding delays to better understand how this affects the physical LEDs.

1. Do not rip up the breadboard! You will build on this for the next part.

unsigned char read\_button(unsigned char row, unsigned char col)

{

unsigned char buttonstate = 0;

//Turn pull-ups on

PORTD = 0xF0;

//Set row-col low

DDRD |= 1<<row;

//We need to wait for this to propagate before reading!

\_delay\_us(10);

if((PIND & (1<<(col+4))) == 0){

\_delay\_ms(20);

if((PIND & (1<<(col+4))) == 0){

buttonstate = 1;

}

}

DDRD &= ~(1<<row);

return buttonstate;

}

int main(void)

{

while(1){

for(unsigned char row = 0; row < 4; row++){

if (read\_button(row, 0)){

set\_led(row);

}

}

}

}

Try pressing the buttons to see which ones are active. This code is only checking a single column, so we will need to expand this to scan all rows and columns.

1. Expand the previous code to scan over all rows. This will be done in a new function defined as follows:

unsigned char decode\_buttons(void)

{

unsigned char row, col;

unsigned char button = 0xff;

for(row = 0; row < 4; row++){

for(col = 0; col < 4; col++){

if(read\_button(row, col)){

button = row | (col << 4);

}

}

}

switch(button){

case (0<<4) | 0: return 0x0F;

case (0<<4) | 1: return 0x0E;

case (0<<4) | 2: return 0x0D;

case (0<<4) | 3: return 0x0C;

case (1<<4) | 0: return 0x0B;

case (1<<4) | 1: return 0x0A;

case (1<<4) | 2: return 0x09;

case (1<<4) | 3: return 0x08;

case (2<<4) | 0: return 0x07;

case (2<<4) | 1: return 0x06;

case (2<<4) | 2: return 0x05;

case (2<<4) | 3: return 0x04;

case (3<<4) | 0: return 0x03;

case (3<<4) | 1: return 0x02;

case (3<<4) | 2: return 0x01;

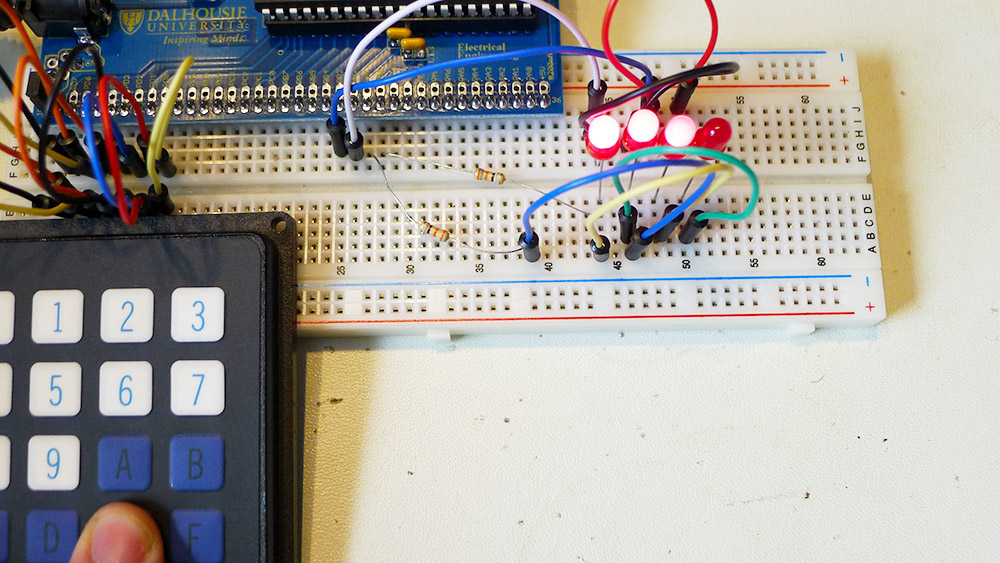
case (3<<4) | 3: return 0x00;

default: return 0xFF;

}

}

1. Scan through the button press, and display the button in the LEDs using binary code. For example this shows what happens if you press the ‘E’ button, which is 1110 in binary:



The following main function will get you started, but you will need to expand this, in particular not all statements were placed into the case statement! Add an example for each button and check they are working.

int main(void)

{

unsigned char button, tempbutton;

while(1){

tempbutton = decode\_buttons();

if (tempbutton != 0xFF){

button = tempbutton;

} else {

\_delay\_ms(30);

}

switch(button){

case 0:

set\_led(0xFF);

break;

case 1:

set\_led(3);

break;

case 2:

set\_led(2);

break;

case 3:

set\_led(3);

\_delay\_ms(50);

set\_led(2);

break;

case 4:

//FILL IN FROM 4 to 0x0D

break;

case 0x0D:

//FILL THIS IN

break;

case 0x0E:

set\_led(0);

\_delay\_ms(50);

set\_led(1);

\_delay\_ms(50);

set\_led(2);

break;

case 0x0F:

//FILL THIS IN

break;

default:

break;

}

}

}

# Lab Questions

1. What causes problems when trying to read a switch, and how do we fix it?
2. The LED multiplexing could only illuminate certain LED patterns, as LEDs were linked together. How could we illuminate an arbitrary pattern (hint: our human eyes will blend together two quickly displayed patterns)?
3. Include code from each section in your report.