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: Raw Read/Write of Characters

## 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

## Background

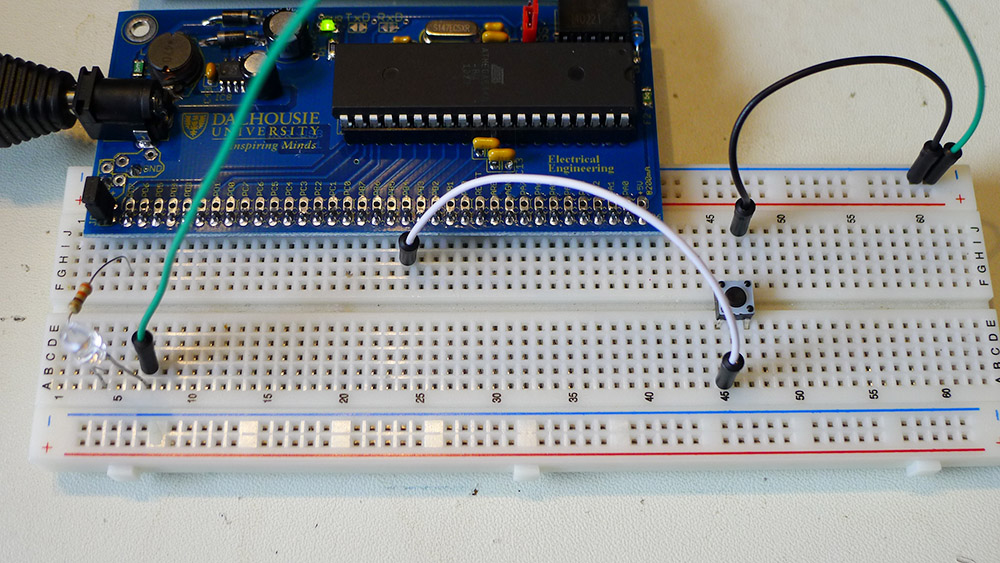
Serial communication without an explicit clock is known as *asynchronous* communication. It requires you to configure a number of settings on both sides of the connection, these settings are the ‘agreement’ between sending and receiving about the meaning of bits.

When debugging your code, this serial communication is an extremely useful tool. It will allow you to print the value of variables, or modify the value of variables without having to recompile your code.

As the first part of this, we will send and receive single characters at a time. This will be extended in Part 2 to send and receive complete strings, and parse those strings into variables.

## Procedure

1. Power up the digital trainer board, and connect the programmer. This lab does not use any external devices (no LEDs, etc).



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

#include <avr/io.h>

#include <util/delay.h>

void init\_uart(void)

{

UCSR0B = (1<<RXEN0) | (1<<TXEN0);

UBRR0 = ????

}

void write\_char(char c)

{

//Wait for UDR0 to be ready for writing

//Write data byte

UDR0 = c;

}

char check\_char(void)

{

//Check if data ready for reading

return (UCSR0A & (1<<RXC0));

}

char read\_char(void)

{

//Wait for data

while((UCSR0A & (1<<RXC0)) == 0);

//Return data

return UDR0;

}

int main(void)

{

init\_uart();

//STEP1: Just print 'A'

while(1){

write\_char('A');

\_delay\_ms(500);

}

//STEP2: Check if character received

while(1){

if (check\_char()){

write\_char('A');

} else {

write\_char('B');

}

\_delay\_ms(500);

}

//STEP3: Full echo

while(1){

write\_char(read\_char());

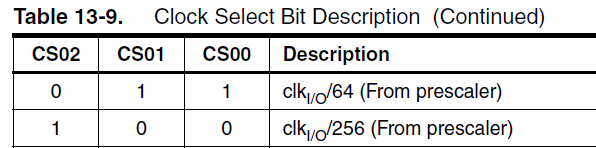
}

}

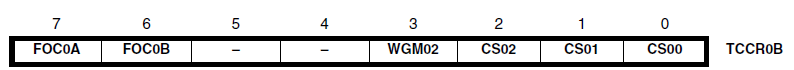
1. Before being able to compile this code, we need to finish a few things. The first is to determine the setup of TCCR0A and TCCR0B. We want the timer to overflow somewhere between 100-5000 times per second.

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!

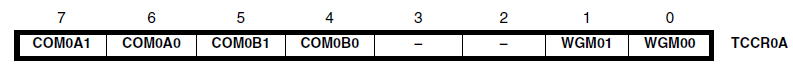
Finding Table 13-9, you can see some prescaler settings:



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:



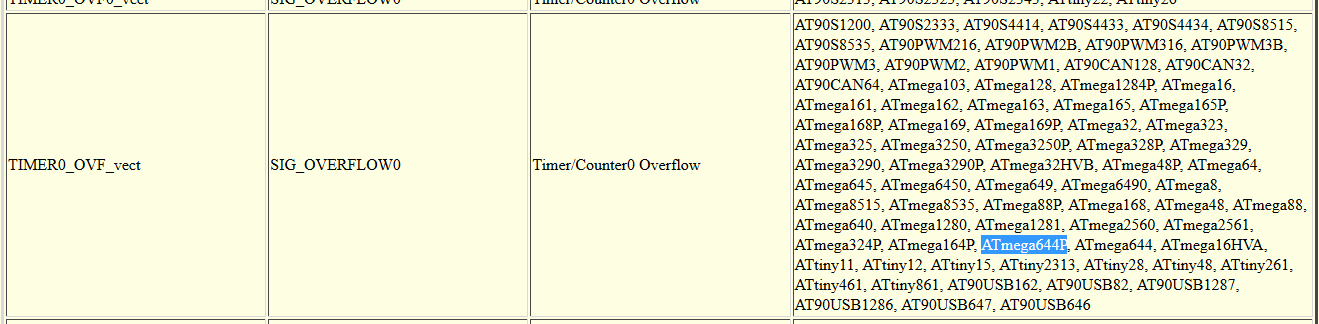
See the datasheet for a description of those bits. This means our setup area of the code looks like this:

//Timer configuration

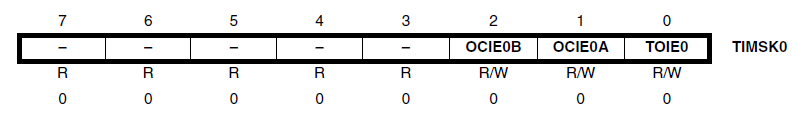
TCCR0A = 0;

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: Using <stdio.h>

## 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);

}

}