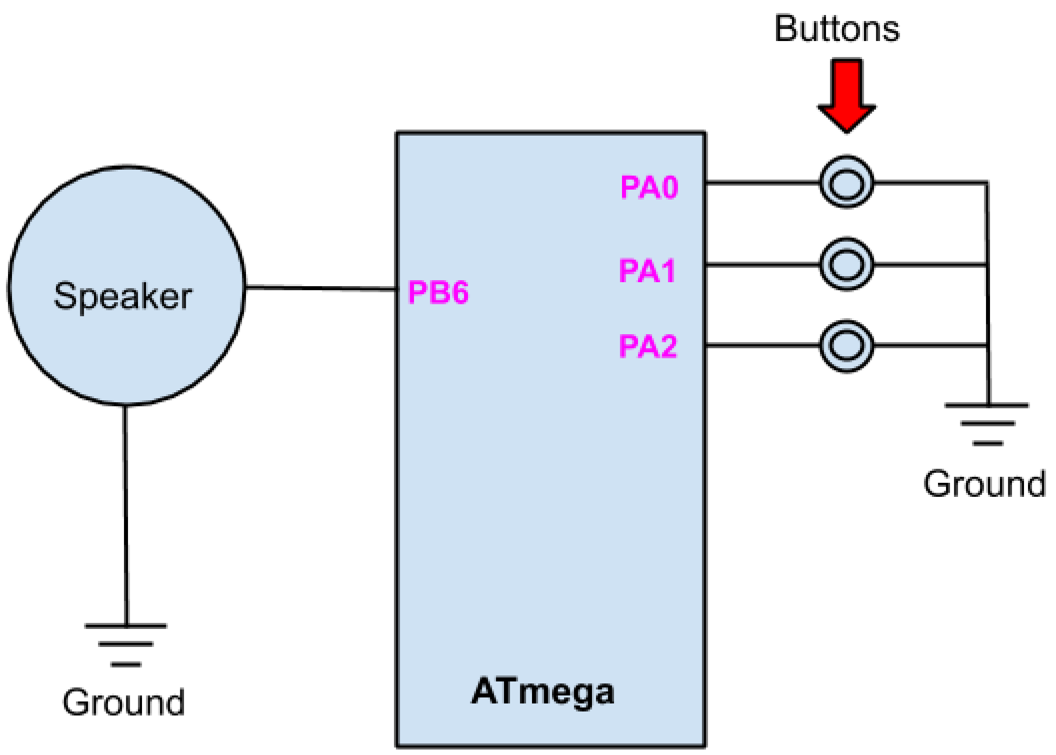
Lab 9: PWM

UCR EE/CS120B

Pre-lab

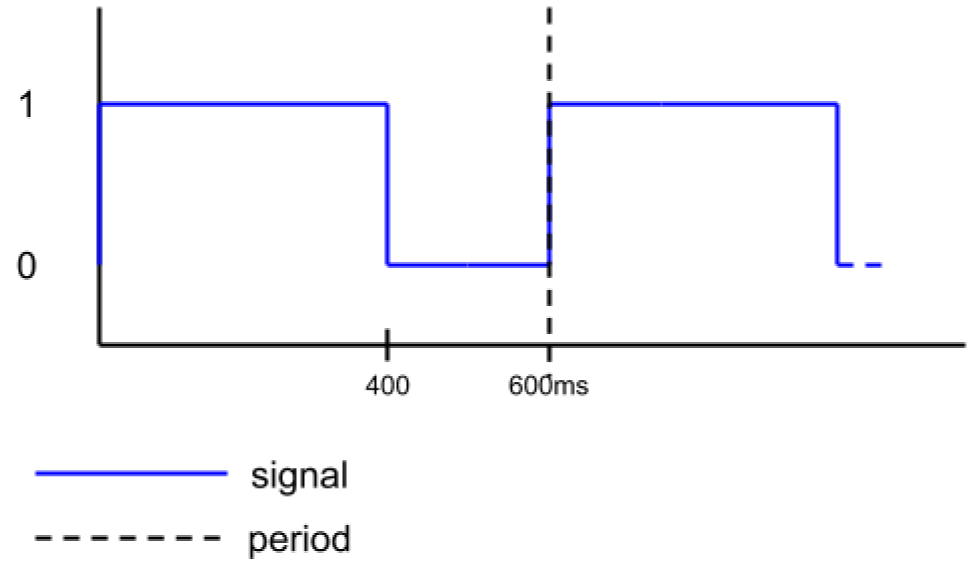
Read the entire lab manual and come prepared with a board wired according to the image below. Placement of the buttons can vary, but the speaker must be connected to PB6. **Note:** Make sure that the pin right below the '+' sign on the speaker is connected to PB6.



Introduction

A pulse width modulator (PWM) is a programmable component that generates pulses to achieve a specified period and duty cycle. Such pulses have many uses, such as generating sounds of a particular frequency, controlling a motor with a specific average input voltage, communicating information, and more. Some key PWM definitions are:

- **Period** (T): The time for one cycle of a repeating pulse pattern. The signal below has a period of 600 ms.
- **Duty cycle**: The percentage of time that a signal is high during the period. The signal below has a duty cycle of 66.6% since the signal is high for 400 ms of the 600 ms period, and 400/600 = 66.6%



This lab uses pulse width modulation to create sounds of various frequencies.

How does the period influence sound from a speaker?

Sound is generated by pulsing a speaker's input voltage at a particular frequency. The higher the frequency, the higher the sound. The range of an average adult's hearing is between 20 Hz and 16,000 Hz. Hz is a measure of frequency, indicating the number of pulses per second. Frequency is the inverse of period: frequency = 1/period.

How does a speaker work?

A speaker contains a magnetic component called a “diaphragm”. When positive voltage is supplied to the speaker, the diaphragm is pulled inwards. When no voltage is supplied, the diaphragm is released and moves outwards. Pulsing the voltage causes the diaphragm to pulse at the same frequency, causing sound waves to be created at that frequency. A 50% duty cycle means voltage is supplied just as often as not, minimizing the delay between the signal's high and low values, maximizing the number of pulses per period. A 50% duty cycle produces the loudest possible tone. If the duty cycle is higher or lower, then less time is spent pulsing, resulting in a quieter tone.

Musical sounds

The frequencies of middle notes of a piano are listed below. Based off of these frequencies, the period can be calculated by inverting the frequency. The following frequencies, and calculated periods, will be used throughout the lab.

Musical Note	Frequency (Hz)
C_4	261.63
D_4	293.66
E_4	329.63
F_4	349.23
G_4	392.00
A_4	440.00
B_4	493.88
C_5	523.25

ATmega1284's PWM functionality

In previous labs, writing a synchSM that could pulse at different frequencies could get extensive and complicated really quick. These synchSMs can be greatly simplified by using the ATmega1284's built in PWM functionality.

The ATmega has multiple methods of implementing PWM. The method used in this lab uses a timer and a counter. A variable is set to a desired 16-bit value. The counter counts up to the value of the variable. When the counter equals the value of the variable, an interrupt flag is set, and the pin outputting the PWM signal is toggled.

The functions below set up `TCCRnA`, `TCCRnB` and `OCRnA` for the `timer/counter 3`. Copy and paste this code to the top of the project. For more information on what these function do, refer to page 140 of the ATmega1284's datasheet.

```
// 0.954 hz is lowest frequency possible with this function,
// based on settings in PWM_on()
// Passing in 0 as the frequency will stop the speaker from generating sound
void set_PWM(double frequency) {
	static double current_frequency; // Keeps track of the currently set frequency
	// Will only update the registers when the frequency changes, otherwise allows
	// music to play uninterrupted.
	if (frequency != current_frequency) {
		if (!frequency) { TCCR3B &= 0x08; } //stops timer/counter
		else { TCCR3B |= 0x03; } // resumes/continues timer/counter

		// prevents OCR3A from overflowing, using prescaler 64
		// 0.954 is smallest frequency that will not result in overflow
		if (frequency < 0.954) { OCR3A = 0xFFFF; }

		// prevents OCR3A from underflowing, using prescaler 64
		// 31250 is largest frequency that will not result in underflow
		else if (frequency > 31250) { OCR3A = 0x0000; }

		// set OCR3A based on desired frequency
		else { OCR3A = (short)(8000000 / (128 * frequency)) - 1; }

		TCNT3 = 0; // resets counter
		current_frequency = frequency; // Updates the current frequency
	}
}

void PWM_on() {
	TCCR3A = (1 << COM3A0);
		// COM3A0: Toggle PB3 on compare match between counter and OCR3A
	TCCR3B = (1 << WGM32) | (1 << CS31) | (1 << CS30);
		// WGM32: When counter (TCNT3) matches OCR3A, reset counter
		// CS31 & CS30: Set a prescaler of 64
	set_PWM(0);
}

void PWM_off() {
	TCCR3A = 0x00;
	TCCR3B = 0x00;
}
```

NOTE: Removing the "`static double current_frequency`" and the "`if (frequency != current_frequency)`" guard will update the frequency every time it is called, even if the frequency passed in is the same as the last time `set_PWM()` was called.

Function Descriptions:

`PWM_on()`: Enables the ATmega1284's PWM functionality.

`PWM_off()`: Disables the ATmega1284's PWM functionality.

`set_PWM(double frequency)`: Sets the frequency output on `OC3A` (Output Compare pin). `OC3A` is pin PB6 on your microcontroller. The function uses the passed in frequency to determine what the value of `OCR3A` should be so the correct frequency will be output on PB6. The equation below shows how that output frequency is calculated, and how the equation in `set_PWM` was derived.

$$f_{\text{OC2}} = f_{\text{Clock}} \;/\; (2 * \text{prescaler} * (\text{OCR3A} + 1))$$

IMPORTANT: PB6 MUST BE USED AS YOUR SPEAKER OUTPUT PIN because the PWM connects to that pin. So, make sure to set PB6 as output (`DDRB = x1xx xxxx;`). In order to program your ATMega you will need to disconnect the speaker since PB6 is used by the programmer.

NOTE: There are multiple pins on the ATMega1284 that can be used for PWM. If you are using a pin other PB6 you will need to set up a different timer. The ATMega 1284 datasheet can help with this endeavor. Don't forget that some of the timer/counters are 16 bit while others are only 8 bits.

Exercises

1. Using the ATmega1284's PWM functionality, design a system that uses three buttons to select one of three tones to be generated on the speaker. When a button is pressed, the tone mapped to it is generated on the speaker. **Criteria:**
    a. Use the tones C_4, D_4, and E_4 from the table in the introduction section.
    b. When a button is pressed and held, the tone mapped to it is generated on the speaker.
    c. When more than one button is pressed simultaneously, the speaker remains silent.
    d. When no buttons are pressed, the speaker remains silent.

Video Demonstration: http://youtu.be/_w4BmDvA9mw

2. Using the ATmega1284's PWM functionality, design a system where the notes: C_4, D, E, F, G, A, B, and C_5, from the table at the top of the lab, can be generated on the speaker by scaling up or down the eight note scale. Three buttons are used to control the system. One button toggles sound on/off. The other two buttons scale up, or down, the eight note scale. **Criteria:**
    a. The system should scale up/down one note per button press.
    b. When scaling down, the system should not scale below a C.
    c. When scaling up, the system should not scale above a C.

Hints:

- Breaking the system into multiple synchSMs could make this part easier.

Video Demonstration: http://youtu.be/BjPZhS_gGzU

3. (**Challenge**) Using the ATmega1284's built in PWM functionality, design a system where a short, five-second melody, is played when a button is pressed. **NOTE:** The melody must be somewhat complex (scaling from C to B is NOT complex). **Criteria:**
    a. When the button is pressed, the melody should play until completion
    b. Pressing the button again in the middle of the melody should do nothing
    c. If the button is pressed and held, when the melody finishes, it should not repeat until the button is released and pressed again

Hints:

- One approach is to use three arrays. One array holds the sequence of notes for the melody. Another array holds the times that each note is held. The final array holds the down times between adjacent notes.

Video Demonstration: http://youtu.be/4fBw0aR3nvQ

Submission

Each student must submit their source files (`.c`) and any new/modified header file through Gradescope according to instructions in the lab submission guidelines.

Don’t forget to commit and push to Github before you logout!