1 Objectives

By the end of this lab, you will:

- Connect a speaker and produce more complex sounds
- Generating Arbitrary Sounds on Arduino

Materials: We will be providing the parts necessary for this lab (other than the Arduino and breadboard).

2 Sampling: Producing more complex sounds

In the lab sound lab, we use Arduino `tone()` function to produce sound at a certain frequency. To produce arbitrary sound waves, we would need to be able to output any analog voltage on a spectrum (0 to 5V, for example) in order to perfectly match our wave. Our Arduino, however, can only output digital signals: HIGH (5V) or LOW (0V). Calling the `tone()` function generates a PWM waveform on a particular pin (ie, a square wave) at a particular frequency. This works well for playing simple sounds consisting of single-frequency tones, but we need to go a step further to produce more complex signals.

To approximate an arbitrary, continuous signal, digital systems represent these signals as a discrete series of samples, which are scalar values that represent the magnitude of the signal at a certain time index. An example of this is shown in Figure 1 with each $S_i$ representing a discrete sample of the analog signal $S(t)$. 
Note that each sample is recorded at a periodic rate, $T$, the inverse of which is called the sampling frequency or **sample rate**. A full discussion of sampling theory is far beyond the scope of this lab. However: for the purpose of reproducing a waveform, this means that we need to output a new sample at every time interval $T$ in order to accurately reconstruct the signal—a hard real-time operation.

Furthermore, it makes intuitive sense that we cannot reproduce waveforms of a frequency higher than our sampling frequency. In fact, to perfectly reconstruct a waveform without losing information, the maximum frequency we can reproduce is equal to **half** the sampling rate—a fundamental theorem in sampling theory by Claude Shannon and Harry Nyquist. You can read more about this here: [https://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem](https://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem)
3 Generating Arbitrary Sounds on the Arduino Uno

Our microcontroller, we can meet our real-time constraint for playing sounds by using interrupts to control when to output each sample in a stored waveform. In this lab, we will configure interrupts to reconstruct sounds with a sampling frequency of 8 kHz, which is high enough to reproduce low-quality audio.

3.1 Using Interrupts

Our Arduino contains three hardware timers (called Timer0, Timer1, and Timer2) that can accurately measure time—these are blocks of hardware separate from the CPU core that operate in an asynchronous manner from code execution. These timers are can be used for timekeeping operations, PWM generation, frequency measurement, and more. These timers can be configured to produce interrupts: when an interrupt occurs, the CPU stops its current operation and runs interrupt service routine, or ISR, which is a function with a special definition you define in your program. After the ISR returns, the CPU restores its previous context and continues running. In this way, we can use the ISR to perform operations that happen at precise time intervals.

Rather than download an external library to configure the timer, we can do it ourselves in a few lines of code that manipulate the hardware configuration registers. Examine the sample code below, which configures interrupts to fire 8000 times per second: the first function defines the ISR we want to run, and the second function configures the timer.

In this example template, the ISR just increments a counter (which you could use for timekeeping). We will extend this ISR slightly to output a sample from a given waveform each time it runs.

If you want to learn more about how the timer configuration works, you can read about the Timer peripherals and their configuration registers in the Atmega328P datasheet—the relevant sections are referenced in the comments for the example.
# define SAMPLE_RATE 8000

volatile unsigned long timer_count = 0;

// This function is the Interrupt Service Routine (ISR), which is called
// every time an interrupt occurs for this timer
ISR(TIMER1_COMPA_vect)
{
    timer_count++; // Example: count number of time intervals elapsed

    // . . . Play a sample here . . .
}

void init_timer(void) // Call this function from setup()
{
    noInterrupts(); // Disable all interrupts

    // Clear Timer1 register configuration
    TCCR1A = 0;
    TCCR1B = 0;

    // Configure Timer1 for CTC mode (WGM = 0b0100)
    bitSet(TCCR1B, WGM12);

    // Disable prescaler
    bitSet(TCCR1B, CS10);

    // Set timer period, which is defined as the number of
    // CPU clock cycles (1/16MHz) between interrupts - 1
    OCR1A = (F_CPU / SAMPLE_RATE) - 1;

    // Enable timer interrupts when timer count reaches value in OCR1A
    bitSet(TIMSK1, OCIE1A);

    interrupts(); // Enable interrupts
}

Task: Copy this example into your code and call the init_timer() function from your setup() function to configure the timer. To demonstrate how to use interrupts: use the provided counter timer_count to blink an LED at a rate of once per second. If your Arduino has a built-in LED (try pin LED_BUILTIN), you can use it for this purpose. Your code should not use any Arduino timing functions such as delay() or millis().

### 3.2 Approximating Analog Sounds

Before we can play sounds, however, we have one additional problem: the Arduino can only output digital signals (HIGH and LOW), it cannot output analog voltages corresponding to the magnitude
of the waveform we want to reproduce.

We can, however, approximate an analog signal using PWM. By varying the duty cycle of the PWM signal, we can approximate an analog signal by transmitting the same amount of power to the speaker per unit time as as the analog waveform we want to reproduce. The accuracy of the approximated signal is directly reliant on the frequency of the PWM output. To make this work, the frequency of the PWM signal must be much higher than the analog wave it is trying to reproduce, as is visible in the diagram in Figure 2.

![Sound wave diagram](https://adafruit.com)

Figure 2: Sound wave. Source: adafruit.com

Previously, we generated PWM waveforms using the `analogWrite()` function. Unfortunately, the Arduino implementation for generating PWM waveforms does not use a high enough PWM frequency for our purposes, so we need to configure another timer to generate PWM signals at a higher frequency.

We have provided source code to configure the timer to do this on the course website in the files `fastpwm.cpp` and `fastpwm.h`. Download these files and add them to your sketch by selecting **Sketch > Add File...** from the Arduino IDE menus. Examine these files briefly to understand the interface provided to you. Most notably, the function `fastpwm_init()` configures Timer2 to generate a PWM signal on **pin 11** at a frequency of about 32 kHz. You can call this function from your sketch to start the PWM generator.

After initialization, you can set the duty cycle of the waveform (0–255) by passing a sample value to the function `fastpwm_play_sample()`.

Some important notes about using the fast PWM method:

- It is important to understand the distinction between how Timer2 and Timer1 are used here: Timer1 (from the previous section) is used to generate periodic interrupts at 8000Hz—you will use the ISR for this timer to handle playing samples and timekeeping operations. Timer2 is only used to generate the PWM signal that we use to approximate sounds. Once configured, your sketch only interacts with it by changing the duty cycle of the signal using `fastpwm_play_sample()`.
- It is the change in PWM duty cycle between samples that produces an audible sound output.
Therefore, calling `fastpwm_play_sample()` only once, or calling it repeatedly with the same value, will appear to have no effect as the frequency of the PWM signal is too high to hear.

- Timer2 is also used by the Arduino `tone()` and `analogWrite()` functions. Therefore, you cannot use these Arduino functions while playing samples with fast PWM, as it will change the timer settings.

For the final portion of this lab, we can again play a simple melody, but this time we can compose it from arrays of samples for three waveforms representing tones at 200, 400, and 800 Hz. These tones will be played at 8000 samples per second with 8 bit resolution.

**200hz (40 samples):** 255, 253, 248, 241, 230, 217, 202, 185, 166, 147, 127, 107, 88, 69, 52, 37, 24, 13, 6, 1, 0, 1, 6, 13, 24, 37, 52, 69, 88, 107, 127, 147, 166, 185, 202, 217, 230, 241, 248, 253

**400hz (20 samples):** 255, 248, 230, 202, 166, 127, 88, 52, 24, 6, 0, 6, 24, 52, 88, 127, 166, 202, 230, 248

**800hz (10 samples):** 255, 230, 166, 88, 24, 0, 24, 88, 166, 230

Save these sequences as arrays in your sketch. You can now use them to generate a simple melody by playing each tone for a certain duration using the Timer1 ISR. For example: to play the 200Hz tone, play one sample from an array of the 200Hz on every timer interrupt (ie, every 1/8000 seconds); when reaching the end of the sample array, continue again from the starting index. Assuming a sampling frequency of 8000 Hz, playing 8000 samples corresponds to 1 second of audio.

**Task:** Use the circuit for the speaker from last lab (either with or without the amplifier) Use the three sample arrays to play a simple melody lasting at least five seconds using interrupts.