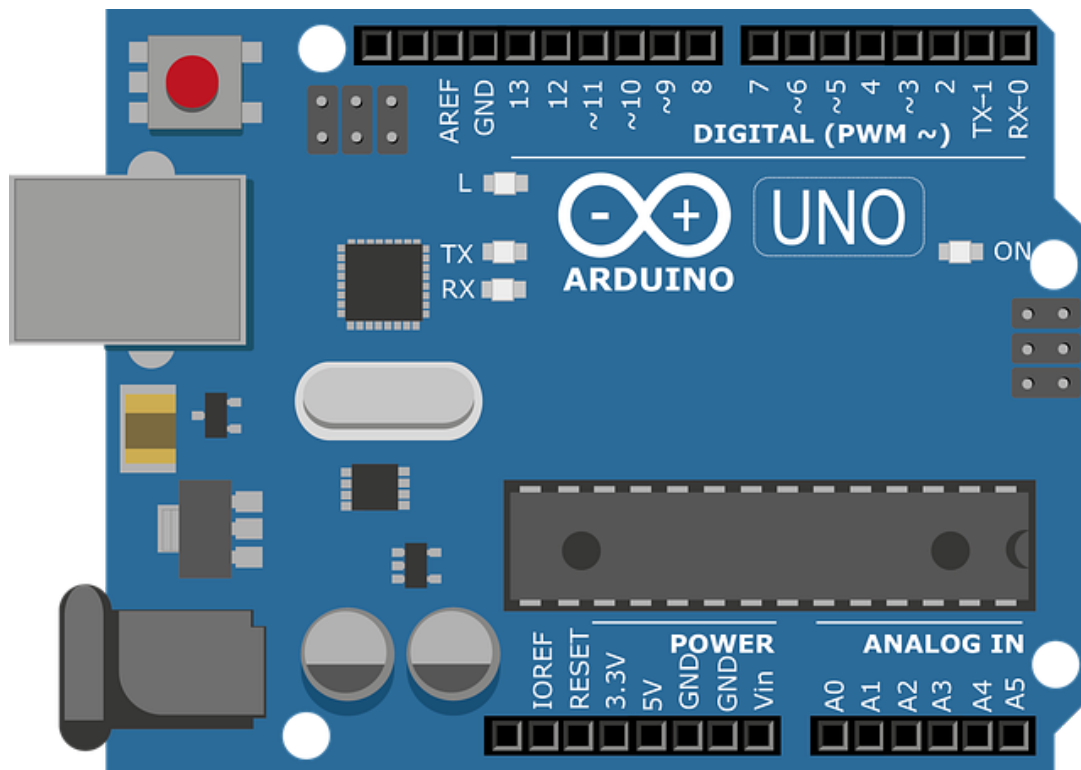


# PRÁCTICAS CON ARDUINO



## Práctica 3: Zumbador

### Grupo de Trabajo

Mónica Cañón Pascual

IES Pino Rueda

Departamento de Tecnología

Abril 2018



## Finalidad de la práctica

1. Esta práctica consiste en controlar un zumbador, controlando las notas..
2. Intentaremos hacer una pequeña canción, la banda sonora de las Guerra de la Galaxias

## Información

La piezoelectricidad es un fenómeno que ocurre en determinados cristales que, al ser sometidos a tensiones mecánicas, adquieren una polarización eléctrica y aparece una diferencia de potencial y cargas eléctricas en su superficie que generan una tensión eléctrica.

Este fenómeno también ocurre a la inversa: se deforman bajo la acción de fuerzas internas al ser sometidos a un campo eléctrico. El efecto piezoeléctrico es normalmente reversible: al dejar de someter los cristales a un voltaje exterior o campo eléctrico, recuperan su forma.

Es decir, que son materiales (el cuarzo es el más conocido) que si los sometemos a una tensión eléctrica variable (como una señal PWM, que ya nos son familiares) vibran.

Es un fenómeno bastante conocido y muchos encendedores domésticos de gas funcionan bajo este principio. Un resorte golpea un cuarzo y como resultado tenemos la chispa que enciende el gas o el calentador de agua con un característico click).

En otro orden de cosas, los circuitos electrónicos digitales, suelen disponer de un reloj interno que vibra a una velocidad patrón, basados en cristales de cuarzo piezoeléctrico. El cristal de Arduino late a 16Mhz por segundo y la flecha indica su posición.

Si conectamos un piezo con una señal digital, vibran a una frecuencia que sigue bastante fielmente la variación eléctrica con que los excita, y si vibran a la frecuencia audible, oiremos el sonido que producen. A un componente que hace esto, le llamamos Buzzer o zumbador.

Naturalmente, la calidad del sonido que producen dista bastante de lo que podríamos denominar alta fidelidad. Pero es suficiente para generar tonos audibles (como la típica alarma de los despertadores digitales) e incluso tonos musicales reconocibles que podemos secuenciar, hasta en piezas musicales (por más que uno quisiera estar en otro lugar cuando las oyes).

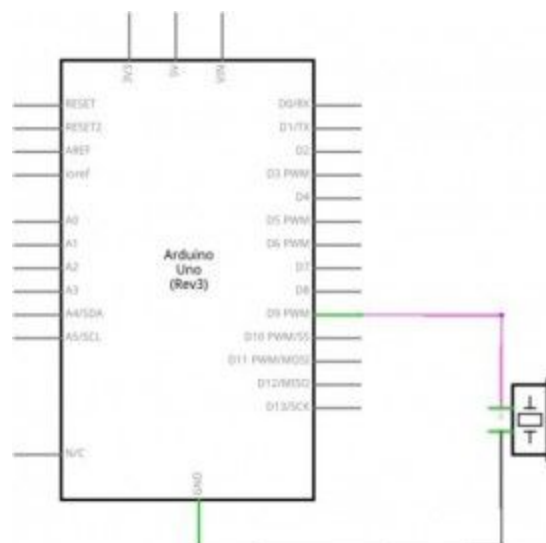


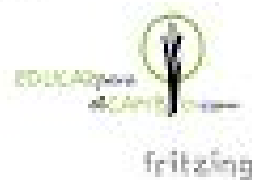
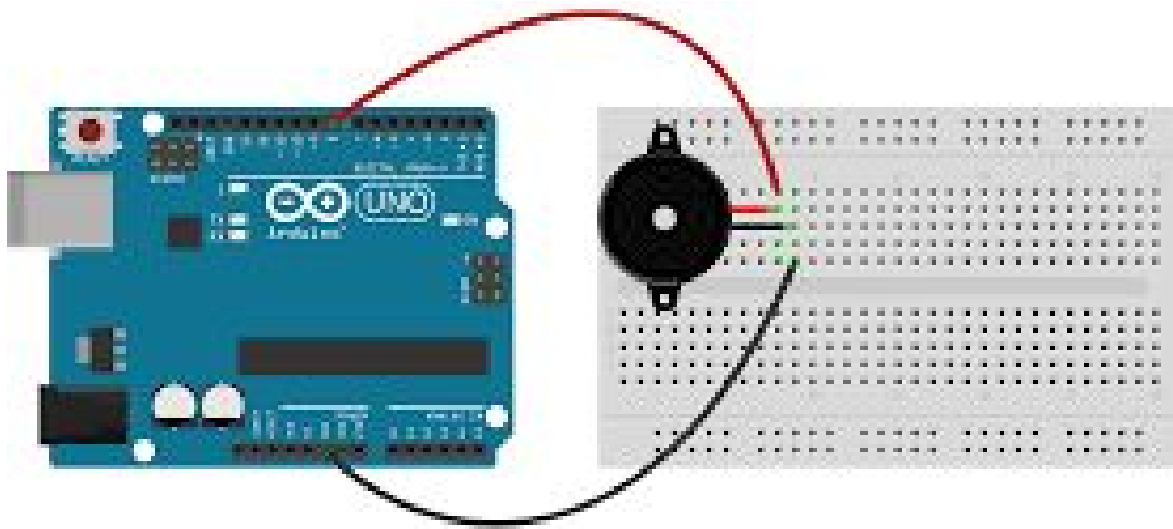
Como antes o después, disponer de una señal acústica en vuestros proyectos, acaba siendo útil, vamos a ver cómo podemos montar estos elementos, y que tipo de opciones tenemos disponibles.

En esta sesión, montaremos un circuito muy sencillo con un zumbador.

## Hardware necesario

### I. Esquema de conexiones



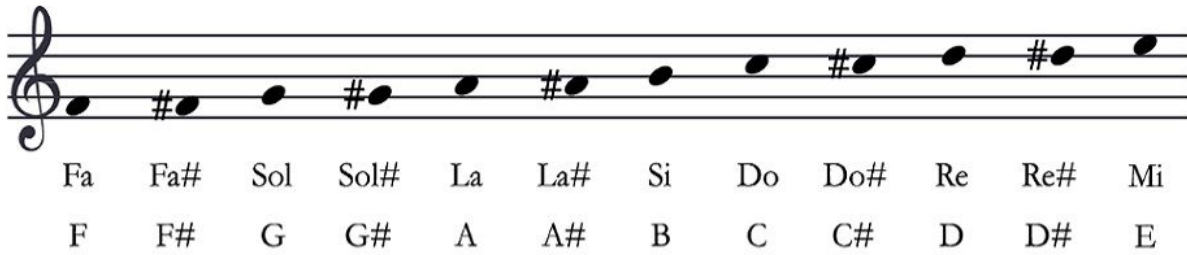


## Programación

Snap4Arduino	IDE arduino
??	<p>Como salida digital:</p> <pre> sketch_oct28a § int zum = 13;  void setup() {   pinMode(13, OUTPUT); }  void loop () {   digitalWrite(13, HIGH);   delay(2000);   digitalWrite(13, LOW);   delay(5000); } </pre>

## Para hacer una pequeña canción

```
/** Included libraries */  
  
/** Global variables and function definition */  
const int zumbador = 11;  
  
/** Setup */  
void setup() {  
  pinMode(zumbador, OUTPUT);  
}  
/** Loop */  
void loop() {  
  tone(zumbador, 329, 1000);  
  delay(1000);  
  tone(zumbador, 349, 1000);  
  delay(1000);  
  tone(zumbador, 329, 1000);  
  delay(1000);  
  tone(zumbador, 349, 1000);  
  delay(1000);  
  tone(zumbador, 392, 2000);  
  delay(2000);  
  tone(zumbador, 329, 1000);  
  delay(1000);  
  tone(zumbador, 349, 1000);  
  delay(1000);  
  tone(zumbador, 329, 1000);  
  delay(1000);  
  tone(zumbador, 349, 1000);  
  delay(1000);  
  tone(zumbador, 392, 2000);  
  delay(2000);  
}
```



a instrucción **tone** cuyos argumentos son el pin de salida al cual tenemos conectado el zumbador, la frecuencia que es la que variará la nota y la duración en milisegundos (opcional).

**tone**(pin, frecuencia)

**tone**(pin, frecuencia, tiempo)

## Actividades y propuestas de mejora

```

/* Star Wars Song Selector
 * ----- Program to choose between two melodies by using a potentiometer and
 * a piezo buzzer.
 * Inspired by:
 * https://code.google.com/p/rbots/source/browse/trunk/StarterKit/Lesson5_PiezoPlayMelody/Lesson5_PiezoPlayMelody.pde
 */

// TONES //
// Defining the relationship between note, period & frequency.

// period is in microsecond so P = 1/f * (1E6)

#define c3 7634
#define d3 6803
#define e3 6061
#define f3 5714
#define g3 5102
#define a3 4545
#define b3 4049

```

```

#define c4 3816 // 261 Hz
#define d4 3401 // 294 Hz
#define e4 3030 // 329 Hz
#define f4 2865 // 349 Hz
#define g4 2551 // 392 Hz
#define a4 2272 // 440 Hz
#define a4s 2146
#define b4 2028 // 493 Hz
#define c5 1912 // 523 Hz
#define d5 1706
#define d5s 1608
#define e5 1517 // 659 Hz
#define f5 1433 // 698 Hz
#define g5 1276
#define a5 1136
#define a5s 1073
#define b5 1012
#define c6 955

#define R 0 // Define a special note, 'R', to represent a rest

// SETUP //

int speakerOut = 9; // Set up speaker on digital pin 7
int potPin = A0; // Set up potentiometer on analogue pin 0.

void setup() {
  pinMode(speakerOut, OUTPUT);
  Serial.begin(9600); // Set serial out if we want debugging
}
//}

// MELODIES and TIMING //
// melody[] is an array of notes, accompanied by beats[],
// which sets each note's relative length (higher #, longer note)

// Melody 1: Star Wars Imperial March
int melody1[] = { a4, R, a4, R, a4, R, f4, R, c5, R, a4, R, f4, R, c5, R, a4, R, e5, R, e5,
R, e5, R, f5, R, c5, R, g5, R, f5, R, c5, R, a4, R};
int beats1[] = { 50, 20, 50, 20, 50, 20, 40, 5, 20, 5, 60, 10, 40, 5, 20, 5, 60, 80, 50, 20,
50, 20, 50, 20, 40, 5, 20, 5, 60, 10, 40, 5, 20, 5, 60, 40};

```

```

// Melody 2: Star Wars Theme
int melody2[] = { f4, f4, f4, a4s, f5, d5s, d5, c5, a5s, f5, d5s, d5, c5, a5s, f5, d5s,
d5, d5s, c5};
int beats2[] = { 21, 21, 21, 128, 128, 21, 21, 21, 128, 64, 21, 21, 21, 128, 64, 21, 21, 21,
128 };

int MAX_COUNT = sizeof(melody1) / 2; // Melody length, for looping.

long tempo = 10000; // Set overall tempo

int pause = 1000; // Set length of pause between notes

int rest_count = 50; // Loop variable to increase Rest length (BLETCHEROUS
HACK; See NOTES)

// Initialize core variables
int toneM = 0;
int beat = 0;
long duration = 0;
int potVal = 0;

// PLAY TONE //
// Pulse the speaker to play a tone for a particular duration
void playTone() {
    long elapsed_time = 0;
    if (toneM > 0) { // if this isn't a Rest beat, while the tone has
        // played less long than 'duration', pulse speaker HIGH and LOW
        while (elapsed_time < duration) {

            digitalWrite(speakerOut, HIGH);
            delayMicroseconds(toneM / 2);

            // DOWN
            digitalWrite(speakerOut, LOW);
            delayMicroseconds(toneM / 2);

            // Keep track of how long we pulsed
            elapsed_time += (toneM);
        }
    }
    else { // Rest beat; loop times delay
        for (int j = 0; j < rest_count; j++) { // See NOTE on rest_count
            delayMicroseconds(duration);
        }
    }
}

```



```

}
}

// LOOP //
void loop() {
  potVal = analogRead(potPin); //Read potentiometer value and store in potVal
  variable
  Serial.println(potVal); // Print potVal in serial monitor

  if (potVal < 511) { // If potVal is less than 511, play Melody1...

    // Set up a counter to pull from melody1[] and beats1[]
    for (int i=0; i<MAX_COUNT; i++) {
      toneM = melody1[i];
      beat = beats1[i];

      duration = beat * tempo; // Set up timing


      playTone(); // A pause between notes
      delayMicroseconds(pause);
    }
  }
  else // ... else play Melody2
  for (int i=0; i<MAX_COUNT; i++) {
    toneM = melody2[i];
    beat = beats2[i];

    duration = beat * tempo; // Set up timing

    playTone(); // A pause between notes
    delayMicroseconds(pause);
  }
}

/*
* NOTES
* The program purports to hold a tone for 'duration' microseconds.
* Lies lies lies! It holds for at least 'duration' microseconds, _plus_
* any overhead created by incrementing elapsed_time (could be in excess of
* 3K microseconds) _plus_ overhead of looping and two digitalWrite()
*
* As a result, a tone of 'duration' plays much more slowly than a rest
* of 'duration.' rest_count creates a loop variable to bring 'rest' beats
* in line with 'tone' beats of the same length.

```



```
*  
* rest_count will be affected by chip architecture and speed, as well as  
* overhead from any program mods. Past behavior is no guarantee of future  
* performance. Your mileage may vary. Light fuse and get away.  
*/
```