

arduinonext

hands-on booklet



1

David Cuartielles
(Arduino Co-Founder)

Arduino 8-bit Sound Generation

David Cuartielles (1974, Zaragoza, Spain) is a Telecommunications Engineer. In 2000 he developed the net-art called *The Wake Interactive*, a web version of *Finnegans Wake*, the James Joyce classic, that led him to K3, the art and communication school at Malmö University, Sweden.



His academic activity is focused on model conceptualization, indoor localization systems and real-time technologies for artistic performances.

Currently he is Head of the Prototyping Laboratory at K3, and owns a Research Fellow position in Interaction Design at the Medea Research Studio.

David has a permanent interest in embedded electronics and education, having taught courses and delivered lectures at several institutions around the world including UCLA, NYU, Samsung Art and Design Institute, Copenhagen Institute for Interaction Design, Tecnológico de Monterrey, and others.

Co-author of the well-known Arduino platform for electronic prototyping since 2005, David has continued developing new boards and teaching tools which expand Arduino's application scope.

As part of his outreach activities David collaborates with Elektor by supplying didactically driven articles.

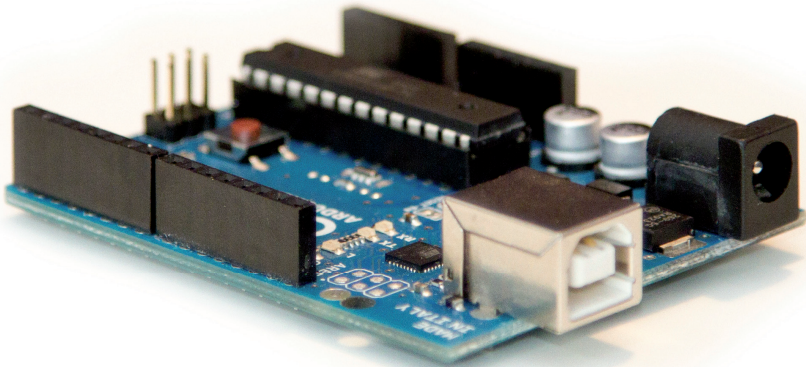
In its ongoing outreach work, David is collaborating with Elektor writing the didactic articles who underpinning this collection.

David Cuartielles
(Arduino Co-Founder)

Arduino 8-bit Sound Generation



Arduino 8-bit sound generation



This booklet is going to lay the foundations for creating Interactive Sound Machines using a standard Arduino Uno board. However, all the knowledge described here as well as the code can be easily ported to Arduino Mega, Arduino Mini or any other member of the Arduino 8-bit family. Jump the Arduino bandwagon, this stuff comes straight from a creator of the platform!

Practice makes perfect and in this the first of a series of booklets I will focus in presenting all the bits and pieces you need to understand in order to play 1-bit sound from a digital pin on your Arduino board. We will start by looking at the easiest way to create sound by means of a simple piezo buzzer or a speaker. I will then introduce the Arduino Tone Library, as a simplified way to achieve the same functionality. And I will close by introducing an advanced technique that allows playing short sounds stored in the form of .wav files.

When it comes to the theory, you will be introduced to a technique known as 1-bit Delta Sigma Digital to Analog conversion, but don't be scared by the name, the methods and technologies are presented along with examples you can easily reproduce with a minimal set of parts.

Get out your materials

If you aim at reproducing all the examples in this booklet you need to have:

- an Arduino Uno board, though some of the other boards in the Arduino line using ATmega328, ATmega168, ATmega128 or ATmega256 will work as well;
- a USB cable to connect your Arduino to a computer;
- a piezo buzzer to play tones;
- alternatively a mini loudspeaker or headphones and a socket to connect them to the Arduino board;
- a computer running the Arduino IDE and a tool called SoundData for the IDE (check the download links in the references section).

Sound machines

When talking about interactive music instruments, I like to refer to them as Sound Machines. They consist of three blocks: the data used to generate the sound, the user interface, and the actual sound engine.

The data refers to the way the sound is used. It could be generated, we could play back sounds from a sample collection, or just perform real-time sound alteration. Data is the actual sound, the tones in a piano, or the bytes stored inside the channels in a sampler.

The user interface defines the way the sound structures will be manipulated, or the mechanisms to modify the sound characteristics. The easiest

example is the volume control usually implemented as potentiometer or rotary encoder. The UI is what the user manipulating the instrument has his/her hands (or mind) on to alter the sound engine's behavior.

Finally, the engine defines the way data is displayed. It takes the data and renders it as sound according to the control parameters introduced by the user through the UI. In our case, the engine is the Digital to Analog converter we are going to implement via software.

From blink to bee

The “Hello World” example to start using Arduino is making the LED on pin 13 blink. The code looks like this:

```
/* Blink
   Get the LED on pin 13 to go on/off
   http://arduino.cc
*/

int ledPin = 13; // define the LEDs pin

void setup()
{
    pinMode( ledPin, OUTPUT ); // configure the pin as output
}

void loop()
{
    digitalWrite( ledPin, HIGH ); // turn the pin on
    delay( 1000 ); // wait 1sec
    digitalWrite( ledPin, LOW ); // turn the pin off
    delay( 1000 ); // wait 1sec
}
```

The example is self explanatory. With little changes we can use the same example to start playing sound out of our Arduino. We will be using a piezoelectric buzzer and later on we will move into using a proper loudspeaker to increase the quality of the sound output.

The piezoelectric buzzer

The contact microphone, also known as piezo buzzer is an electronic component made of a combination of two discs of different materials. One of

them is metallic and the other one is usually ceramic, having the piezoelectric properties. When applying a voltage to the component, the materials will repel each other, producing an audible click. Making the voltage difference zero will cause the materials to return to their original position, again producing a click sound.

By applying a voltage of a sufficiently high frequency, the click produced by the piezoelectric will modulate into an audible tone.

You can test this by connecting a piezoelectric buzzer to Arduino by wiring its positive pin to, say, Arduino's pin 8 and the negative pin to Arduino ground (GND), see **Figure 1**. Take the previous code and modify the line that determines the pin the LED is connected to:

```
int ledPin = 8; // define the pin for your Speaker or Piezoelectric
```

When running this program you will hear the piezoelectric clicking once per second, which is the times when the voltage changes at its pins. You can modify the code to make the delay between clicks. The smaller the delay, the more the clicks will merge into a modulated tone. Try to change both lines in your program affecting the time between clicks to be:

```
delay( 1 );
```

Now you will be hearing a sound with a frequency of 500 Hz. You will also notice the sound to be louder due to a property of the piezoelectric components. They resonate in the kilohertz range, at least the ones we find commercially available, as they are designed to be buzzers or contact microphones.

If you want to experiment further in sound production with this technique, I would encourage you to not use the `delay()` function from Arduino. The reason for this is that you will need a better time-resolution to produce a richer selection of tones. The function `delayMicroseconds()` is much more suitable to sound production, as it is three orders of magnitude more accurate.

Adding all these changes to the original Blink program, will give us the Bee program:


```
/* Bee
   Make a piezoelectric oscillate on pin 8
   http://arduino.cc
*/

int piezoPin = 8; // define where the piezoelectric is connected

void setup()
{
    pinMode( piezoPin, OUTPUT);
}

void loop()
{
    digitalWrite( piezoPin, HIGH );
    delayMicroseconds( 1000 );
    digitalWrite( piezoPin, LOW );
    delayMicroseconds( 1000 );
}
```

Playing tones

According to Fourier's Law every sound is the result of adding a series of sinusoidal sound sources having different phases and amplitudes. We could say that sine and cosine waves are the fundamental components of sound. Unfortunately microcontrollers, and the one on the Arduino Uno is not an exception, cannot imitate the sinusoidal shape perfectly. We can anyway produce square waves by repeatedly switching a pin HIGH and LOW. The tones produced in this way have the same frequency, but are not clean as they add components to the pure sinewave. In musical terms, the instrument making square waves has a different tonal characteristic than the one making sinusoidal ones, although the tones proper are the same. The only issue we find here is that sound is expressed in frequency (hertz or Hz) while microcontrollers work with time. The total amount of time a HIGH-LOW oscillation lasts is what we call the period (seconds). There is a relationship between period p and frequency f as expressed in the following formula:

$$f = \frac{1}{p} \rightarrow p = \frac{1}{f}$$

In other words, period is the inverse of frequency and vice-versa. In this way, if we want to know the delay needed to play say, the A_4 tone using Arduino, we need to do a bit of maths again like:

$$p = \frac{1}{440} = 0.002272 \text{ s} = 2.272 \text{ ms} = 2272 \text{ } \mu\text{s}$$

If we want the program “Bee” to play A_4 , we need to modify it so that the total time in both calls to `delayMicroseconds()` adds up to 2272 microseconds.

```
digitalWrite( piezoPin, HIGH );
delayMicroseconds( 1136 );
digitalWrite( piezoPin, LOW );
delayMicroseconds( 1136 );
```

Table 1. Available musical notes / tone frequencies

| Tone | Frequency [Hz] | Period [μs] | Delay [μs] |
|-------|----------------|--------------------------|-------------------------|
| C_4 | 261.63 | 3822 | 1911 |
| D_4 | 293.66 | 3405 | 1703 |
| E_4 | 329.63 | 3024 | 1517 |
| F_4 | 349.23 | 2863 | 1432 |
| G_4 | 392.00 | 2551 | 1276 |
| A_4 | 440.00 | 2272 | 1136 |
| B_4 | 493.88 | 2025 | 1012 |
| C_5 | 523.25 | 1911 | 956 |

The different tones within a scale can be mapped to time-delays so that they can be played using Arduino. **Table 1** shows a full octave.

Arduino’s tone library

There is more to sound playing than just the tone. Musical scores are expressed in notes, which are the tones played for just a certain amount of time. In order to simplify the possibility of playing basic melodies using Arduino, we added a library to the system that handles all the math explained so far in this article, as well as note durations. Now that you understand how to play a tone at a relatively low level, I think it is convenient to introduce this abstraction to make easier for you to create programs playing sound.

This library is called `tone` and brings in a function allowing playing a tone. In the background this is done by controlling some of the internal timers in the processor. One timer takes care of the tone itself, while the other monitors its duration. There are two functions to play sound, called the same, but with a different amount of parameters.

```
tone( pin, frequency );           // play a tone
tone( pin, frequency, duration ); // play a note, duration in milliseconds
```

You should note how time is measured. The function's argument 'duration' will play the note for a certain number of milliseconds (ms). If you want to have a function counting the time in a way that is closer to how it is made in musical scores, you will have to decide how long the different notes will be. The following code assigns durations to the different notes considering that the basic note lasts 0.5 seconds.

```
/* we define durations as numbers between 1 and 7:
 1 - whole note - 1 unit
 2 - half note - 0.5 units
 3 - crotchet - 0.25 units
 4 - quaver - 0.125 units
 5 - semi quaver - 0.0625 units
 6 - demi semi quaver - 0.03125 units
 7 - hemi demi semi quaver - 0.015625 units
*/
void playNote( int speaker, int theTone, int duration )
{
  // we give for granted that the half note lasts 0.5 segundos
  long time = 500 / pow( 2, duration-1 );

  // assign the note to the speaker
  tone( speaker, theTone, time );
}
```

The function above could also be written in a way that would make use of the more common variable of the beats per minute (bpm). In that way we could make the instrument play at different speeds depending on the desired tempo for the melody.

```

/* we define durations as numbers between 1 and 7:
   [...]
*/
void playNote( int speaker, int theTone, int duration, int bpm )
{
  // menmotecnic: 120 bpm - - > 1000 milliseconds for half note
  // source http://bradthemad.org/guitar/tempo_calculator.php
  long time = ( 1000 * bpm / 120 ) / pow( 2, duration );

  // assign the note to the speaker
  tone( speaker, theTone, time );
}

```

Chained melodies

It’s “awesome and stuff” to play a short melody at some point during the execution of a program. A melody consists of notes and silences. Here we are going to see how to play a melody stored as an array of numbers. Inside the Arduino IDE there is an example dealing exactly with this. Open the program under *File* → *Examples* → *2.Digital* → *toneMelody*:

```

/*
  Melody

  Plays a melody
  [...]
*/
#include "pitches.h"

// notes in the melody:
int melody[] = {
  NOTE_C4, NOTE_G3, NOTE_G3, NOTE_A3, NOTE_G3, 0, NOTE_B3, NOTE_C4};

// note durations: 4 = quarter note, 8 = eighth note, etc.:
int noteDurations[] = { 4, 8, 8, 4, 4, 4, 4, 4 };

void setup() {
  // iterate over the notes of the melody:
  for( int thisNote = 0; thisNote < 8; thisNote++ ) {

    // to calculate the note duration, take one second
    // divided by the note type.
    //e.g. quarter note = 1000 / 4, eighth note = 1000/8, etc.
    int noteDuration = 1000 / noteDurations[thisNote];
    tone( 8, melody[thisNote], noteDuration );
  }
}

```

```

// to distinguish the notes, set a minimum time between them.
// the note's duration + 30% seems to work well:
int pauseBetweenNotes = noteDuration * 1.30;
delay( pauseBetweenNotes );

// stop the tone of playing:
noTone( 8 );
}
}

void loop() {
// no need to repeat the melody.
}

```

You will notice that the melody is stored in two arrays: the notes are in one, while the note durations are in a different one. Durations are expressed differently from what we saw earlier. Our previous example was using durations expressed in the way it is done in music.

The first command in the program includes a file called `pitches.h` that comes as part of the code into a different tab within the example. The file sadly is too large for printing here. It includes a series of constants representing the frequencies for each tone. In that way, the constant `NOTE_A4` represents the numeric value 440, or the 440 Hz for the note A_4 .

```

/*****
*                               *
*           Public Constants     *
*                               *
*****/

#define NOTE_B0  31
#define NOTE_C1  33
#define NOTE_CS1 35
[...]
#define NOTE_G4  392
#define NOTE_GS4 415
#define NOTE_A4  440
#define NOTE_AS4 466
[...]
#define NOTE_CS8 4435
#define NOTE_D8  4699
#define NOTE_DS8 4978

```

The above example can be modified to make use of the beats-per-minute (BPM), what will allow changing the speed at which the melody plays. We

just need to include the `playNote` function we made earlier and modify the way we express the durations to be expressed in the same terms as in our new function:

```
#include "pitches.h"

int melody[] = {NOTE_C4, NOTE_G3, NOTE_G3, NOTE_A3, NOTE_G3, 0, NOTE_B3,
NOTE_C4};
int noteDurations[] = { 2, 3, 3, 2, 2, 2, 2, 2 };
int bpm = 120;

void playNote( int speaker, int theTone, int duration, int bpm )
{
    // mnemonic: 120 bpm - - > 1000
    // 1000 milliseconds for half note
    // source http://bradthemad.org/
    // guitar/tempo_calculator.php
    long time = ( 1000 / bpm * 120 ) / pow( 2, duration );

    tone( speaker, theTone, time ); // assign the note to the speaker
    delay( time*1.30 ); // add 30% for the silence between notes
}

void setup() {
    for( int thisNote = 0; thisNote < 8; thisNote++ ) {
        playNote( 8, melody[thisNote], noteDurations[thisNote], bpm );
    }
}

void loop() {
    // let's listen to it just once
}
```

Now try including a potentiometer or any other analog sensor to modify the BPM rate and in that way play the melody at different speeds.

1-bit Sound Generation ... What?!?

Up to this point, you have been experimenting with code that can be generated directly out of the Arduino language. Now we proceed with methods of hacking the processor on the Arduino Uno (the ATmega 328) at a low level, aiming to create a block of code that will use sound data coming from a WAV file for storage inside program memory — and playing back of course.

The method used is not trivial. A good way to start is to look at sound based in terms of its spectrum rather than from any sort of representation in time.

Sound spectrum

The sound spectrum is a representation of the energy transmitted by, say, a loudspeaker as a function of frequency. Looking at the spectrum of sound, you do not normally get a clear view of the sound itself as in **Figure 1**. Rather, the graph represents the amount of energy for different discrete frequencies within a certain amount of time. Typically we represent the spectrum for a whole song (see **Figure 2**).

Alternatively, we can look at the spectrum of just 0.5 seconds of a song. The smaller the time frame we're watching, the better the spectral representation of the sound generated at that very instant. The shape of the wave generating that spectrum can be anything. As a matter of fact, two different sound signals can generate very similar spectra.

Thus, the size of the time frame determines the similarity of that signal is to the original one. The human ear is stimulated by the energy content of the sound, therefore two signals having identical spectra will be perceived as the same, but only if the time resolution is small enough. This is key to the whole science of sound compression: the ability of getting signals that are 'good enough' for us to understand the sound, even if the sound is very different from the original one.

This is also the way 1-bit sound generation works [1]. We can

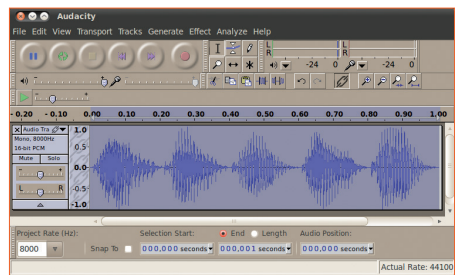


Figure 1. Recording of human speech saying “ta-te-ti-to-tu”.

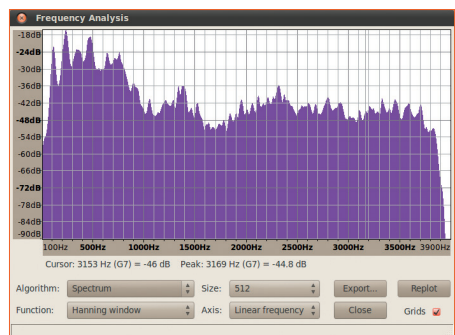


Figure 2. Spectrum of human speech saying “ta-te-ti-to-tu”.

generate sound by having a pulse width modulated (PWM) signal whose average energy level is similar enough to the one of the original signal. This mathematical trick allows generating medium-quality sound by a microcontroller. The following paragraphs show how to take advantage of this. We will start from a WAV file that can be recorded with your computer. Next, we'll filter it, and transform it into an array of numbers for storing inside Arduino.

Optimal digitalization and filtering

Many different tools exist for recording sounds. I can only recommend the use of Audacity [2], an open source and free software tool that provides most of the options needed to reproduce sound with microcontrollers (**Figures 1** and **2** are screenshots from Audacity).

Before you move on, you should filter the sound. I use a low-pass filter with a 4 kHz roll-off frequency. The Arduino sound player shown here uses a sampling frequency of 8 kHz (i.e. 8000 samples per second), which means that if there were sound components above 4 kHz in your original file, you would hear artifacts in the sound.

With microcontrollers it is possible to reproduce reasonable quality sound, however these chips are limited in terms of memory space. Consequently you need to use sound formats of lower quality that will allow generating several seconds of sound without the use of external memory chips.

A sound format that can be of great use is the 8-bit PCM WAVE (also called Microsoft unsigned 8 bit). This sound format is of sufficient quality to reproduce, for instance, recorded human voice. Thus, the file you'll need in the following step of the process should be exported as: mono (1-channel), 8-bit, PCM WAVE.

Converting sound to text

Let's look at importing your sound file as a header file that you can add to your Arduino sketch. Since the ATmega 328 microcontroller has 32 KB of Flash memory space, you can use part of that space to store the sound file. You can store large chunks of data into memory arrays by using the `Progmem` library from Atmel's toolchain. 8-bit sound is nothing but a stream of numbers between 0 and 255. It can be declared like:


```
const unsigned char sounddata_data[] PROGMEM = {128, 128, 128,
[...],
69, 62, 59, 57, 52, 50, 56, 65, 74, 86, 96, 109, 116, };
```

I have created a tool for Arduino's IDE that enables you to open WAV files and import them directly as part of your code. You can download it from the link mentioned in the reference list, and add it to your sketchbook folder. Place the file into your Arduino sketchbook folder and uncompress it there.

It will add the following folder to your sketchbook: `tools/SoundData`. After rebooting the IDE, you will see a new item under the Tools menu called `SoundData`. Clicking on it produces a dialog window enabling you to select a WAV file (see **Figure 3**). The second button, titled `Generate Code` will open the WAV file, check whether it is properly encoded, and add a new tab to your code titled `sounddata.h`. This new file contains everything you need to play your WAV file as a 1-bit sound. The file will look like this:

```
// soundData for your Arduino sound project
// automatically generated data
// - original sound file:/development/tmp/matis.wav
// - sampleRate:8000
// - encoding:8 bits
// - channels:1

const int sounddata_length = 7969;

const signed char sounddata_data[] PROGMEM = {127, 127, 127, 127, 127, 127,
[...],
69, 62, 59, 57, 52, 50, 56, 65, 74, 86, 96, 109, 116, };
```

But keep on reading before pressing the `Generate Code` button on the dialog window, because there is more to it!

The Sound Player

Playing back a sampled sound is not obvious, since it requires changing some of the features of

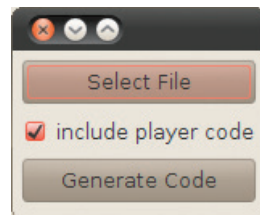


Figure 3. Dialog Window to import WAV files into your Arduino sketches as header files.

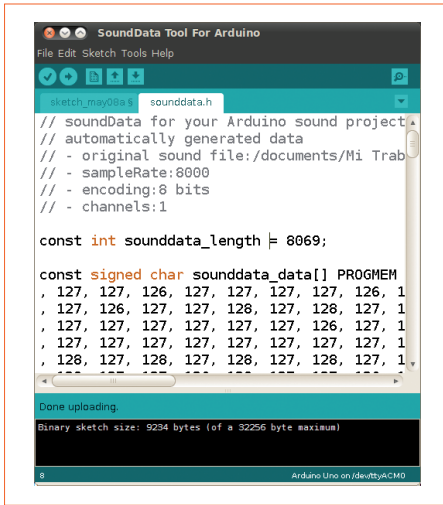


Figure 4. View of the “ta-te-ti-to-tu” sound file within the Arduino IDE after importing it with the SoundData tool.

the Arduino core works. There exist libraries for Arduino Uno that hide all of the complexity of making this type of sound player. However, I want you to take a chance and see how this is done at low level, like in **Figure 4**.

The trick that can get Arduino to play a sound sample is the so-called Fast PWM, a feature of Atmel’s Atmega family (other brands have it as well, but Arduino Uno runs on Atmel chips). There is a register that allows running PWM at the amazing rate of up to half the clock speed. This allows nice things to be done like playing sound with 1-bit outputs. The only limitation of Fast PWM is that it operates at a resolution

of 8 bits only. That’s why you should encode your sound files at 8 bits.

To get it all to work, you use two out of the three internal timer registries inside the processor:

- The first clock operates at SAMPLE_RATE and is used to read the next value from the computer generated sounddata_data array. This value sets up the duty cycle for the PWM stream being output through pin 11.
- The second timer ticks at 62500 times per second (16000000 / 256), i.e. much faster than the playback rate (default SAMPLE_RATE is 8 kHz). As explained earlier, this generates a digital signal whose spectrum is very similar in shape to that of the original sound file. In other words, by means of increasing the PWM frequency, the generated signal gets a spectrum that resembles the one of the sound. In many applications this is more than enough to play sound at a low price.

Note: by adding this code to your Arduino sketch you will be overriding some of the basic commands of the Arduino language. E.g. the PWM will stop working at some of the pins, and the delay() function will not operate as expected. This will only affect this sketch.

The first time this technique was shown in the Arduino world was back in 2008. Michael Smith published an example on the Arduino playground that would play the sound of a MAC computer booting up, upon reset of the Arduino Diecimila board. Michael's code was based upon the work of many others which I have included as references [3].

Just to make it easy, the SoundData tool will not just generate the sound information, but also include the code to the basic sound-playback example inside your sketch. Beware, make sure your sketch is empty when calling the tool, as it will overwrite anything on your IDE.

The basic program to play 1-bit sound (**Figure 5**) includes a lot of low level commands in order to override the timers. I will describe the functions one by one. Let's start with `TIMER1`:

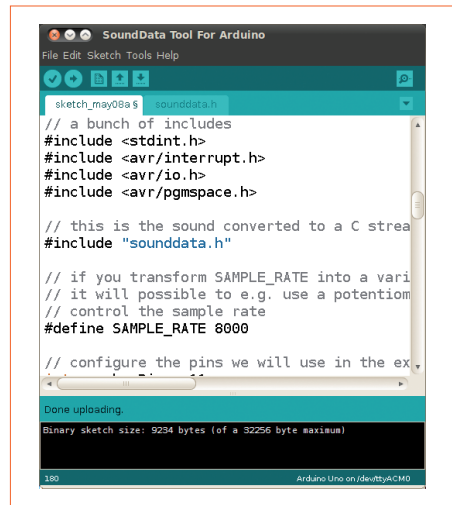


Figure 5. Screenshot of the basic sound player after automatically generating the code.

```
ISR(TIMER1_COMPA_vect) {
    if (sample >= sounddata_length) {
        if (sample == sounddata_length + lastSample) {
            // this is the condition of reaching the last sample
            stopPlayback();
        } else {
            // Ramp down to zero to reduce the click at the end of
            // playback.
            OCR2A = sounddata_length + lastSample - sample;
        }
    }
    else {
        // OCR2A is the register in memory that will push
        // PWM at high frequency to pin 11
        // pgm_read_byte reads data out arrays stored in
        // program memory
        OCR2A = pgm_read_byte(&sounddata_data[sample]);
    }

    // increase the sample count
    ++sample;
}
```

ISR is the name for the interrupt handling function inside the microcontroller. An interrupt is an event that will tell the chip to stop doing anything it is into at the time and attend a certain event. Processors can have both internal and external interrupts. The internal ones are timers, while the external ones happen when certain pins change from HIGH to LOW or vice versa.

This one instance of the ISR function is taking care of the arrival of internal TIMER1 events. Every time TIMER1 ticks, this function will do the following:

- increase the counter used to address the sound data;
- check whether the end of the sound data array is reached;
- if not at the end, load the next sample from the array;
- if at the end, fade the sound to zero.

Both timers TIMER1 and TIMER2 are initialized as part of the start-Playback function. Let's see how what looks like:

```
void startPlayback()
{
    pinMode(speakerPin, OUTPUT);

    // Set up Timer 2 to do pulse width modulation on the speaker
    // pin.

    // Use internal clock (datasheet p.160)
    ASSR &= ~(_BV(EXCLK) | _BV(AS2));

    // Set fast PWM mode (p.157)
    TCCR2A |= _BV(WGM21) | _BV(WGM20);
    TCCR2B &= ~_BV(WGM22);

    // Do non-inverting PWM on pin OC2A (p.155)
    // On the Arduino this is pin 11.
    TCCR2A = (TCCR2A | _BV(COM2A1)) & ~_BV(COM2A0);
    TCCR2A &= ~(_BV(COM2B1) | _BV(COM2B0));

    // No prescaler (p.158)
    TCCR2B = (TCCR2B & ~(_BV(CS12) | _BV(CS11))) | _BV(CS10);

    // Set initial pulse width to the first sample.
    OCR2A = pgm_read_byte(&sounddata_data[0]);

    // Set up Timer 1 to send a sample every interrupt.
    cli();

    // Set CTC mode (Clear Timer on Compare Match) (p.133)
    // Have to set OCR1A *after*, otherwise it gets reset to 0!
    TCCR1B = (TCCR1B & ~_BV(WGM13)) | _BV(WGM12);
    TCCR1A = TCCR1A & ~(_BV(WGM11) | _BV(WGM10));
```

```

// No prescaler (p.134)
TCCR1B = (TCCR1B & ~(_BV(CS12) | _BV(CS11))) | _BV(CS10);

// Set the compare register (OCR1A).
// OCR1A is a 16-bit register, so we have to do this with
// interrupts disabled to be safe.
OCR1A = F_CPU / SAMPLE_RATE;    // 16e6 / 8000 = 2000

// Enable interrupt when TCNT1 == OCR1A (p.136)
TIMSK1 |= _BV(OCIE1A);

lastSample = pgm_read_byte(&sounddata_data[sounddata_length-1]);
sample = 0;
sei();
}

```

Although the sequence of low level commands is explained in the code, a summary of it may be useful for the non-expert:

- turn the pin for the speaker into an output;
- configure the board to use the internal clock for this;
- initialise Fast PWM mode;
- configure TIMER2 to run the PWM that will play the sound, the way to set the duty cycle to the PWM is by changing the value of the register called OCR2A;
- load the first sound sample into OCR2A;
- stop the interrupts for a second – `cli()` – so that we can configure the TIMER1 without breaks;
- configure TIMER1 to tick for picking up the next sample;
- load the last sound sample;
- restart the interrupts – `sei()` .

In a similar fashion, you need to have a function that will stop the timers of counting this way once the end of the sound is reached.

```

void stopPlayback()
{
    // Disable playback per-sample interrupt.
    TIMSK1 &= ~_BV(OCIE1A);

    // Disable the per-sample timer completely.
    TCCR1B &= ~_BV(CS10);

    // Disable the PWM timer.
    TCCR2B &= ~_BV(CS10);

    digitalWrite(speakerPin, LOW);
}

```

This leaves you with a series of functions you can call anywhere from a program to play sound this way. In this case, the example by default will call `startPlayback()` within `setup`. In this way the sound will be played just once.

```
void setup()
{
  startPlayback();
}

void loop()
{
  // do nothing
}
```

Closing words

This article is an introduction to different ways of producing sound using Arduino. You have seen how to play tones by means of basic Arduino functions. Libraries have been described that simplify the way basic

melodies can be played using inexpensive piezo buzzers. Finally you got a sneak peek at the programming behind Fast PWM to generate 1-bit sound.

All code chunks and tools discussed here are packed into a ZIP file [4], including properly formatted sound files for you to try all the examples. I also created a new tool for the Arduino IDE that will help you with the importing of short WAVE files into Arduino's program memory. The tool enables you to load sounds and play them back, change their sample rate, play them backwards or scratch the sound. In terms of hardware, you connect your loudspeaker as shown in **Figure 6**.

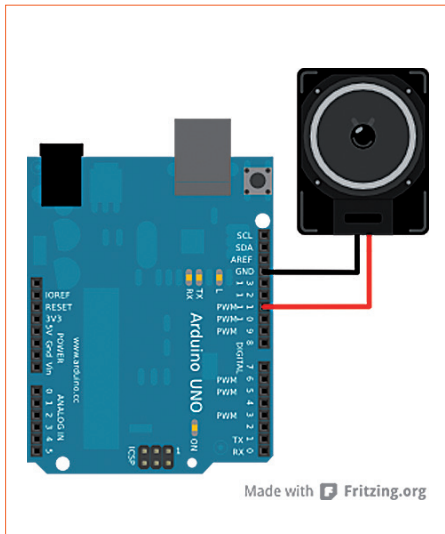


Figure 6. Arduino Uno is perfectly capable of driving a small loudspeaker directly.

But don't stop here! There is a lot to explore — for instance using pin 3 in parallel with pin 11 to produce stereo sound. Or create 8-bit synthesizers with the ability of mixing four sound lines on a single channel. What about transforming your Arduino board into a MIDI activated soundcard.

References

- [1] 1-bit Sigma Delta DA Converters: www.digitalsignallabs.com/presentation.pdf
- [2] Audacity, the free and open source sound studio: audacity.sourceforge.net
- [3] Various references from the example by Michael Smith:
 - Arduino reference on the use of the tone library: arduino.cc/en/Reference/Tone
 - Original article on the Arduino Playground: arduino.cc/playground/Code/PC-MAudio
 - www.uchobby.com/index.php/2007/11/11/arduino-sound-part-1/
 - www.atmel.com/dyn/resources/prod_documents/doc2542.pdf
 - www.evilmadscientist.com/article.php/avrdac
 - <http://gonium.net/md/2006/12/27/i-will-think-before-i-code/>
 - <http://fly.cc.fer.hr/GDM/articles/sndmus/speaker2.html>
 - www.gamedev.net/reference/articles/article442.asp
- [4] Example files for this article: www.elektor.com/120427

**Stay tuned to our communication channels!
Follow us on Facebook and on Twitter.**

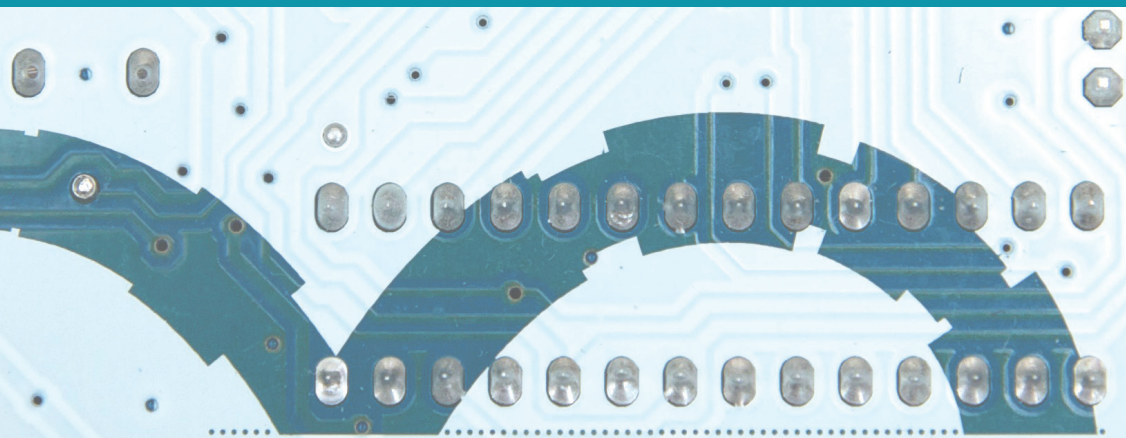


www.facebook.com/arduino-next



[@arduino-next](https://twitter.com/arduino-next)

**Check out the Arduino products already
on sale at www.elektor.com**



Arduinonext is an initiative powered by an electronics and microcontrollers specialist team aiming to help all those who are entering in the technology world, using the well-known Arduino platform to take the next step in electronics.

We strive to bring you the necessary knowledge and experience for developing your own electronics applications; interacting with environment; measuring physical parameters; processing them and performing the necessary control actions.

This is the first title in the 'Hands-On' series in which Arduino platform co-founder, David Cuartielles, introduces board programming, and demonstrates the making of an **8-bit Sound Generator**.

Powered By

