Tone output

Arduino experiment	130135-EN	*Keywords*	tone(), blocking vs. non-blocking functions, arrays	
Version	2018-07-05 / HS	*Level*	Basic course, module 6	*p.* 1/4

What you learn:

- Make a square wave on an output pin
- Understand blocking and non-blocking functions
- Use arrays

- Use a passive piezo buzzer

1 – Connect a piezo buzzer on a breadboard

Different types of buzzers

Start mounting the buzzer as shown – note the unconnected wire.

In this project, we will use the piezo buzzer as a loudspeaker. There are two kinds of buzzers, and only one of these can be used for this purpose. So you are advised to test the buzzer before you go ahead:

Connect one pin of the buzzer to GND and connect a wire to the other pin (as shown). Observe if there is a polarity marking (+ or -). Connect the wire briefly to 5 V. If the buzzer buzzes or beeps it is a so-called active buzzer which cannot be used. If it emits a small tick or rattle it is probably OK.

After the test, the wire is connected to one of the numbered Arduino pins (for instance pin 9).

Write the complete program in the setup() function

To be able to hear when the howling starts and stops we will not write the bulk of the program in `loop()`, like we use to do. By writing everything in `setup()` the lines are executed only once. Enter and test:

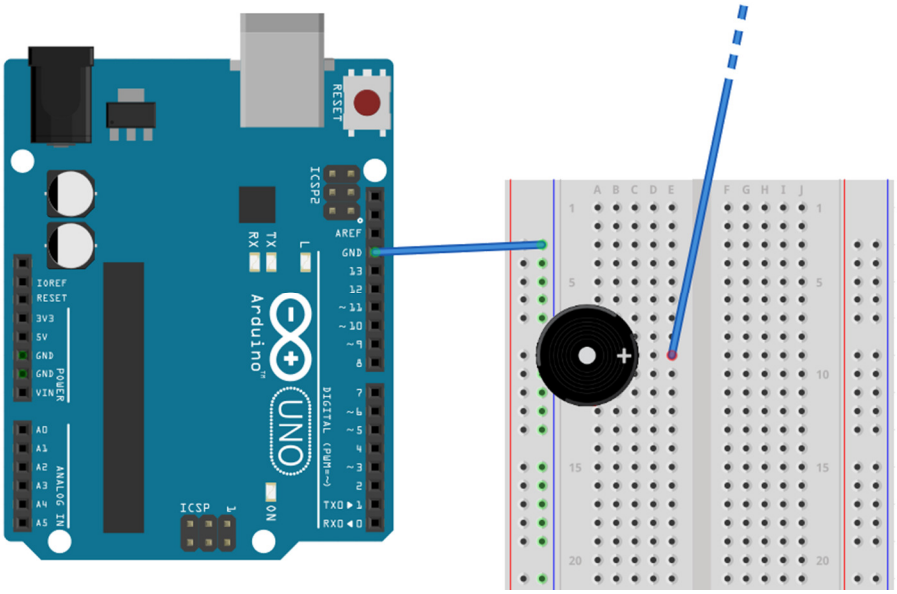
```
const byte pinOut = 9;

void setup() {
  tone(pinOut,440,250); // Concert pitch (440 Hz) a quarter of a second (250 ms).
}

void loop() {            // Nothing here
}
```

Note the lack of configuration of the output pin. It is not needed.

When you have finished the initial test, connect the wire to Arduino pin 9.



Challenge 1

To produce a major triad, you can use the frequencies 440 Hz, 554 Hz, 659 Hz and 880 Hz.

Save the program. Try adding three extra lines with `tone()` calls to make the program play a triad.

The program should play four notes with increasing frequency.

Is it working as you thought? You may want to go back to the original program and compare the sound.

2 – Blocking and non-blocking function calls

Why doesn't it work?

Most of the functions we have used in the previous modules, start (naturally) when they are called – and don't return control to the program until they are finished. A good example is `delay()` – exactly nothing is happening while we're waiting. This type of functions is called *blocking*; the function blocks further program execution until it reaches its end.

It is possible to write functions that starts a given task, but then returns immediately after they have been called – after which the task continues in the background. This is called a *non-blocking* function.

Arduino's `tone()` function is non-blocking. The first three lines in your program each only begins – after which they are interrupted by the *next* call of `tone()` . Only the last call "survives".

This is a wonderfully clever feature in many situation – but right here, it is unwanted.

Write a blocking function that plays tones

This is not especially elegant, but it works: Add a line with `delay(300);` after each call of `tone()` in the program. Test it. It should work…

The extra 50 ms pause for each tone makes it easier to distinguish the individual tones.

OK – we're getting a grip on the problem – time to add a little structure to the program. Instead of calling two different functions for each tone, we will write a new function `play()` which then should call `tone()` and `delay()`.

We will continue working with frequencies, but the milliseconds should be replaced by something more "musical". When working with notes, you specify their duration as fractions of a whole note – here we chose to count in 16ths. This is to be transformed into milliseconds by the function we are writing.

Somewhere in the program we define the conversion factor as a constant – the value 100 is suitable.

```
const int sixteenthsToMillis = 100;
```

Now, the duration of a whole note can be specified as 16, a half note as 8, etc.

Challenge 2

Write a function `void play(int freq, int sixteenths)` that plays a tone which is `sixteenths` long and has a frequency of `freq` . The duration must be adjustable by changing `sixteenthsToMillis` to another value.

When called as shown below, the result must be a major triad (with a tiny delay between the tones).

```
play(440, 2);
play(554, 2);
play(659, 2);
play(880, 6);
```

(If you possess an extremely good sense of rhythm, you *may* feel that the best results are achieved by making the combined duration of tone + tiny delay correspond to the specified duration. In practice, this can be done by using the calculated duration directly in `delay()` while `tone()` is called with a duration that is e.g. 50 ms shorter.)

3 – Arrays

If we should want to play a longer melody with the Arduino, quite a few lines are needed. It would give a better overview if we could just write the tones in a long row (similarly for the durations).

We will now introduce a way of handling this type of data.

Up till now, the numbers we have used in programs have been placed in variables, each having a specific name. By using a data structure called an *array*, a row of numbers is addressed using one name and an *index*.

See toolbox to the right.

This is just what we're looking for…

In the code snippet shown below, two arrays are defined with resp. the frequencies and the corresponding duration for a number of tones.

There is also a variable `notes`, giving the length of the two arrays. Note that it is given as `sizeof(times)`, which works because `times` is an array of *bytes*. There is a point here: If you later wish to add more notes and durations, this variable is automatically updated with the new length.

Toolbox: Arrays

Example:

```
const int fr[] = {440,554,659,880};
```

This defines an *array* of integers with the length 4. The individual array elements are numbered 0 to 3, this number is called the *index* of the element.

For instance `fr[2]` has the value 659.

Example:

```
long a[7];
```

This reserves space for an array of `long` (4 byte integers) of length 7. The elements have undefined initial values.

Example:

```
int x = sizeof(fr);
int y = sizeof(a);
```

After these lines, `x` has the value 8 (there are 4 elements of 2 bytes), while `y` has the value 28 (i.e. 7 elements of 4 bytes).

If each element takes up 1 byte, `sizeof()` returns the length of the array.

The individual elements in the two arrays are identified by specifying the index of the element in **[** square brackets **]**. Numbering starts at 0.

The value of `freq[0]` is 294. The value of `times[7]` is 6.

```
const int freq[] =   {294,294,294,392,392,440,440,588,494,392};
const byte times[] = {  1,  3,  1,  4,  4,  4,  4,  6,  2,  3};
const byte notes = sizeof(times);
```

Arrays are handled well in a `for` loop with a loop variable `i` that can be used as index. In this example, the loop should start with `i=0` and end with `i=notes-1`. This means that the loop condition is `i<notes`.

Challenge 3

Complete the program. The tones in the `freq` array should be played with the corresponding durations in the `times` array. Use a `for` loop for this.

4 – More details about tone( )

These are a few more random details about this function…

One of the ideas behind making `tone()` a non-blocking function is to be able to emit sound signals without interrupting the remaining tasks of the Arduino. You can for instance send out a warning beep and continue measuring, calculating and controlling different outputs during the time that the beep lasts.

You can call `tone()` with only one parameter (the frequency). This will make the tone sound until stopped by a call of the function `noTone()`.

When you use `tone()`, you cannot simultaneously use pins 3 or 11 for PWM signals. (See 130125-EN Arduino as a dimmer.)

5 – A musical doorbell

Having come this far, you are now able to create one of the modern world's technological miracles: The musical doorbell ☺

You need to add a pushbutton switch and to configure the pin you will use in `setup()`.

The melody should not start when the Arduino is turned on, so you got to move the playing away from the `setup()` function.

In `loop()`, you need an `if` construction that checks if the button has been pushed. As soon as it is, the small melody should play.

Challenge 4

Based on the ideas above, write the program for the musical doorbell.

When finished, it should behave like this: When the button is pressed and released, the melody should play once. If the button is kept down, the melody should repeat until the button is released and the melody is over.