# ECS512 Sound Design - Lab 2: Beep!

Required resources:

• Code and sound examples contained in the **Lab2-resources.zip** file – available on QM+.

• **Google Chrome** browser.

• **Atom** – a free on-line text and source code editor.

• **atom-live-server package** – a free add-on for Atom, which will launch a development http server.

• **Sonic Visualiser** – a free analysis and visualization software package developed at QMUL.

This lab builds on the first lab, Introduction to the Web Audio API, by introducing more advanced concepts, and building more functional applications.

The lab is about creating a Beep, with adjustable timing, and recording and analysing the result.

Topics covered in this and other labs are explained in more detail at <https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API> , and links therein. I encourage you to check it out since it is a thorough guide, will reinforce everything being demonstrated here today and will give you a more grounded understanding for future assignments.

## Assignment

*First, do Part 1. Beep and Part 2. Recording an Audio Node below.*

**Beep questions**

Volume.gain.setValueAtTime(0,now) first sets the gain parameter value to 0. So every time the button is clicked, it ramps up from 0 to 1, and then back down to 0.

1. What happens when the button is clicked again before the ramping is finished?

*It resets the value to 0 and starts the ramp again*

1. Now remove this line where the value is set to 0. How does this sound different from before, when the button is clicked again before ramping is finished?

*It doesn't reset the value now, but instead sets a new ramp to 1 from whatever the current value is.*

1. Replace both calls to linearRampToValueAtTime with exponentialRampToValueAtTime. Change the value parameter of linearRampToValueAtTime to 0.001 and for the second call of exponentialRampToValueAtTime, change the end value of the ramp from 0 to 0.001.
   1. How does this change the sound? Which version sounds more natural?

*The sound now appears to ramp up and down more naturally.*

* 1. What sort of error do you get if you don't change the end value from 0 to 0.001 in the second exponentialRampToValueAtTime?

*The error should be something like*

*Beep.js:14 Uncaught RangeError: Failed to execute 'exponentialRampToValueAtTime' on 'AudioParam': The float target value provided (0) should not be in the range (-1.40130e-45, 1.40130e-45).*

*at HTMLButtonElement.triggerBeep.onclick*

1. (optional) Create shortcut keys to trigger the Beep and move the slider up and down. See <https://medium.com/@melwinalm/crcreating-keyboard-shortcuts-in-javascripteating-keyboard-shortcuts-in-javascript-763ca19beb9e> for examples of how to do this.

**Recording audio questions**

1. Compare the two approaches to recording audio, recorder.js or . Which approach do you think is easier to use, and why?

Most people should find Recorder.js easier. Its less code and less use of things outside the Web Audio API in the code. However, it requires the Recorder.js package.

1. Use Recorder.js to add the option to record audio from Part 1 Beep (we use Recorder.js rather than the method from step 7 because MediaRecorder does not support saving as a wav file). Save output using both linearRampToValueAtTime and exponentialRampToValueAtTime (see Question 3). Load these in Sonic Visualiser (the lab demonstrator can help on this), and save the waveforms, spectrum and spectrogram for each one. Explain what you see in the visualization, especially the differences between exponential ramp and linear ramp.

*One should see the ramps in both the waveform and spectrogram.*

1. Now try using setTargetAtTime instead of the exponential or linear ramps, see <https://developer.mozilla.org/en-US/docs/Web/API/AudioParam/setTargetAtTime> . Note that both the current value when it is called, and the target value should never be zero, so have them both as low values like 0.001 instead. As in the previous question, record the audio, save visualiizations and explain what you observe.

*One should see a new form of ramp in both the waveform and spectrogram.*

Using the online submission system, *submit a ZIP archive* containing:

• **Commented** **Code** created during the assignment.

• **Audio files** that you recorded in questions 6 and 7

• **PDF Report**, 3 or 4 sides of A4 total, giving your answers to each question, and including the plots you generated in questions 6 and 7.

## Part 1. Beep

1. Download lab2-resources.zip from QMPlus, unzip it and save it where you want to store your code.

2. Make sure you have the Atom text and source code editor, that you’ve installed the atom-live-server add-on, and that you have installed Chrome and set it to be your default browser.

3. In Atom, from the menu go to File -> Open Folder and select the ‘Beep’ folder from within your unzipped ‘lab2-resources’

4. Create a simple html file, with 4 user interface elements; a ‘Beep’ button, a Frequency range (slider) control, an Attack range control, and a Decay range control. The Attack and Decay are in milliseconds, so could range from 1 to 1000.

Your code will look something like this;

<button id="triggerBeep">Beep</button>

<p>Frequency</p>

<input type="range" min=0 max=8000 value=440 id="Frequency">

<span id="FrequencyLabel"></span>

<p>Attack</p>

<input type="range" min=0 max=1000 value=50 step=0.01 id="Attack">

<span id="AttackLabel"></span>

<p>Decay</p>

<input type="range" min=0 max=1000 value=50 step=0.01 id="Decay">

<span id="DecayLabel"></span>

**beep.html**

5. Now create a separate javascript file with callbacks showing the values of the interface elements.

triggerBeep.onclick = function() { }

Frequency.oninput = function() { FrequencyLabel.innerHTML = this.value + ' Hz'}

Attack.oninput = function() { AttackLabel.innerHTML = this.value + ' ms'}

Decay.oninput = function() { DecayLabel.innerHTML = this.value + ' ms'}

**beep.js**

and include this javascript file by adding the line <script src="Beep.js"></script> near the end of the html file. We’ve kept the Javascript file as small as possible. It uses the element values directly, rather than defining new variables based on the elements.

Try moving the sliders on the interface and see the displayed slider values update.

6. Add a square wave using an Oscillator Node, but connect it to a Gain Node with gain set to 0. We do this rather than not starting it or disconnecting it, since in the next step we will *slowly ramp up* the gain, rather than just connect or start it.

So this part of the Javascript file will look like;

var context = new AudioContext();//Define audio context

var gain = new GainNode(this.context,{gain:0});

var osc = new OscillatorNode(this.context,{type:'square'});

osc.connect(gain);

gain.connect(context.destination);

osc.start();

Note that we could have defined the gain and osc nodes on one line and set parameters on the next, but this is more compact.

7. Now we introduce parameter scheduling. When the Beep button is clicked, we want to ramp up the gain from 0 to 1, and back down again.

This is the callback function for the Beep button.

triggerBeep.onclick = function() {

if (context.state === 'suspended') context.resume()

let now = context.currentTime;

osc.frequency.value = Frequency.value;

Volume.gain.setValueAtTime(0, now);

Volume.gain.linearRampToValueAtTime(1, now + Attack.value/1000);

Volume.gain.linearRampToValueAtTime(0, now + Attack.value/1000 + Decay.value/1000);

}

A few things to point out here.

* Chrome often defaults the audio context to ‘suspended’, so that it won’t produce sound. There are a variety of reasons for this, but mainly it’s because users tend not to want a blast of sound when they go to a web page, unless they request it. So the context needs to be resumed after a user gesture on the page, like clicking the mouse on a button. That’s what if (context.state === 'suspended') context.resume does. The *if* part of the statement is not essential, but it prevents unnecessarily resuming the context for later button clicks.
* linearRampToValueAtTime takes the time in seconds as one of the inputs. The Attack and Decay were given in milliseconds on the interface, so we need to divide them by 1,000.
* We don’t need to specify a start time for linearRampToValueAtTime just the end time. This is because the change starts at the time specified for the previous event.

## Part 2. Recording an audio node

The Web Audio API’s MediaStreamAudioDestinationNode provides the functionality for recording audio from any node in the audio graph, using the MediaRecorder in the MediaStream Recording API

* <https://developer.mozilla.org/en-US/docs/Web/API/MediaStreamAudioDestinationNode>
* <https://developer.mozilla.org/en-US/docs/Web/API/MediaRecorder>

But there’s also an open source package, recorder.js, with similar functionality. Lets see if we can compare them both.

1. Go to <https://github.com/mattdiamond/Recorderjs> and read through the Readme.md on that page. Download the sourcecode using the ‘clone or download’ button. Note: this code is no longer being maintained, but its still widely used, and available (often in slightly modified forms) in other packages.

2. From the downloaded source code, put the file recorder.js in a new folder RecordAudioNode. Note that we’ve included code in the lab resources folder, but you should try typing it in yourself.

3. Open RecordAudioNode in Atom, and create a new file there called RecorderExample.html .

4. This file will only be used to test recording audio, so we’ll start with an extremely simple bit of code to just have audio output from a node. Put the following code in RecorderExample.html.

<script>

var audio\_context = new AudioContext

var Tone= audio\_context.createOscillator()

Tone.start()

</script>

It doesn’t get any simpler than this- just one node, creating a sine wave, not even connected to a destination.

5. Now we want to load recorder.js, set up recording on that node and put in a couple of buttons for starting and stopping a recording, and a place for us to put in recordings.

<script src="recorder.js"></script>

<button onclick="Start()">Start</button>

<button onclick="Stop()">Stop</button>

<audio controls></audio>

<script>

var context = new AudioContext()

var Tone = context.createOscillator()

Tone.start()

var recorder = new Recorder(Tone)

function Start() {

}

function Stop() {

}

</script>

The code is still a shell, and does not do anything. But we can now add all the functionality, mainly by calling make calls to methods on the recorder object.

* recorder.record() – starts capturing audio
* recorder.stop() - stops capturing audio. Subsequent calls to record will add to the current recording.
* recorder.exportWAV() – creates a Blob object (basically a file of raw data) containing the recording in WAV format. It requires a callback function as its argument.
* rec.clear() – clears the recording.

6. Now add a few lines to create a recording using the Start and Stop buttons.

<script src="recorder.js"></script>

<button onclick="Start()">Start</button>

<button onclick="Stop()">Stop</button>

<audio controls></audio>

<script>

var context = new AudioContext()

var Tone = context.createOscillator()

Tone.start()

var recorder = new Recorder(Tone)

function Start() {

context.resume()

recorder.record()

}

function Stop() {

recorder.stop()

recorder.exportWAV((blob) =>

document.querySelector("audio").src = URL.createObjectURL(blob) )

}

</script>

This should work fine, and will record an oscillator.

8. Now create a new file, WAARecorderExample.html, with the code in the example at <https://developer.mozilla.org/docs/Web/API/AudioContext/createMediaStreamDestination> . This is another way to record audio from audio nodes. You may want to search online for information about some of the javascript methods used here, like querySelector and addEventListener.