Assignment 3: Beat-Box

- May be done **individually** or in **pairs**.
- Do not give your work to another student, do not copy code found online without citing it, and do not post questions about the assignment online.
  - Post general questions (not containing your work being marked) to the **CourSys forum**.
  - For personal questions about your work, email: `cmpt-433-help@sfu.ca`
- Submit deliverables to CourSys: https://courses.cs.sfu.ca/
- See the marking guide for details on how each part will be marked.

1. Drum-Beat Info

Your task is to create an application that plays a drum-beat. For this, you'll need a basic understanding of what goes into a drum-beat and music.

Music is played at a certain speed, called the tempo. This tempo is usually in beats per minute (BPM), and often ranges between ~60 (slow) and ~200 (fast) BPM. The beat is the time of a single standard note (called a quarter note).

The “notes” in a drum-beat correspond to the drummer striking different drums (or in our case, playing back recordings of those drums). Often, the music calls for hitting a drum faster than just on the full beats, and hence often notes are played on half-beat increments (called an eight note).

For our standard rock drum beat, we'll be using three drum sounds: the base drum (lowest sound), the snare (the sharp, middle sound), and the hi-hat (high metallic “ting”).

Music is often laid out in measures of 4 beats (hence the “quarter note”). A standard rock beat, laid out in terms of beats, is:

Beat (count from 1)	Action(s) at this time
1	Hi-hat, Base
1.5	Hi-hat
2	Hi-hat, Snare
2.5	Hi-hat
3	Hi-hat, Base
3.5	Hi-hat
4	Hi-hat, Snare
4.5	Hi-hat

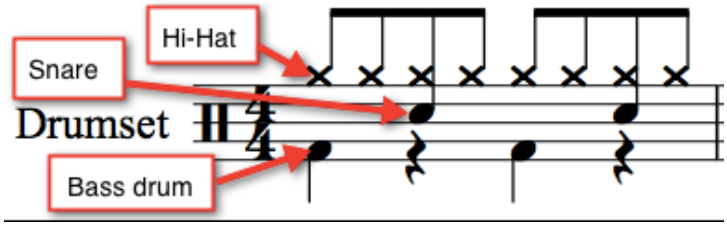

Figure 1: Musical score showing a rock beat.

If you were coding this, you might have a loop that continuously repeats. Each pass through the loop corresponds to a ½ beat (which is an eighth note, and one row in the above table). The loop first plays any needed sound(s) and then waits for the duration of half a beat time.

The amount of time to wait for half a beat is:

Time For Half Beat [sec] = 60 [sec/min] / BPM / 2 [half-beats per beat]

If you want the delay in milliseconds, multiply by 1,000.

2. Folder Structure

You will be submitting a single ZIP file containing your beat-box C/C++ code, your wave files, plus your NodeJS code. When we extract this ZIP file it must have a single makefile which builds and deploys the application, wave files, and the NodeJS code in one command (using just `make` with no arguments). Specifically, your makefile must:

- Build your C/C++ application to a file name `beatbox` deployed to:
  `~/cmpt433/public/myApps/`
- Copy your audio files to:
  `~/cmpt433/public/myApps/beatbox-wav-files/`
- Copy your NodeJS server to:
  `~/cmpt433/public/myApps/beatbox-server-copy/`

You can make no assumptions about either the current user's name (i.e., not likely *your* name, or where we will unzip your code, so don't use relative paths to get to the above locations; use `$(HOME)` instead.

When we run your application on the target, you may assume that:

- We will have correctly loaded the I2C and audio virtual capes.
- We will have run `npm install` in the `~/cmpt433/public/myApps/beatbox-server-copy/` folder either from the host, or the target (makes no difference).
- We have a folder `$(HOME)/cmpt433/public/asound_lib_BBB/` containing the software floating point `libasound.so` library from the BBB.

You may find the following Makefile targets to be useful:

```
wav:
	mkdir -p $(PUBDIR)/beatbox-wav-files/
	cp -R beatbox-wave-files/* $(PUBDIR)/beatbox-wav-files/
node:
	mkdir -p $(PUBDIR)/beatbox-server-copy/
	cp -R as3-server/* $(PUBDIR)/beatbox-server-copy/
```

3. Beat-Box

You will create a Beat-Box application which can play different drum-beats on the BeagleBone using the Zen cape for audio output, and its joystick for input.

3.1 Audio Generation

The application must:

- Generate audio in real-time from a C or C++ program using the ALSA API[1], and play that audio through the Zen cape's head-phone output.
  - Audio playback must be smooth, quite consistent, and with low latency (low delay between asking to play a sound and the sound playing).
  - At times, two sounds will need to be played simultaneously. The program must add together PCM values to generate the sound.
- Generate at *least* the following three different drum beats ("modes"). You may optionally generate more.
  1. No drum beat (i.e., beat turned off)
  2. Standard rock drum beat, as described in section 1.
  3. Some other drum beat of your choosing (must be at least noticeably different). This beat need not be a well-known beat (you can make it up). It may (if you want) use timing other than eighth notes.
  - You may add additional drum beats if you like! Have fun with it!
  - Must use at *least* three different drum/percussion sounds (need not use the ones provided, but should use reasonably well known percussion sounds like a drum, bell, cymbal, …). For example, a rock beat using the base drum, hi-hat, and snare.
- Control the beat's tempo (in beats-per-minute) in range [40, 300] BPM; default 120 BPM. (See next section for how to control each of these).
- Control the output volume in range [0, 100] (inclusive), default 80.
- Play additional drum sounds, on command.

Hints:

- Follow the audio guide on the course website for getting a C program to generate sound.
- Look at the `audioMixer_template.h/.c` for suggested code on how to go about creating the real-time PCM audio playback of sounds.
  - You don't *need* to use this code, and you may change any of it you like.
- For the drum-beat audio clips, you may want to use:
  - base drum: `100051__menegass__gui-drum-bd-hard.wav`
  - hi-hat: `100053__menegass__gui-drum-cc.wav`
  - snare: `100059__menegass__gui-drum-snare-soft.wav`

1 Must get special permission to generate sound using other approaches or frameworks.

3.2 Zen Cape Input Controls

3.2.1 Joystick Requirements

- Press **in** (centre) to cycle through the beats (modes).
  - Default is the standard rock beat, and it should then cycle through the custom beat(s), and then loop back around to none (off), and next back to the standard rock beat again.
  - Must be debounced such that it reliably only switches the mode once per normal user's press on the button.
  - There is no required behaviour for if the user presses and holds the joystick in; you may make it either do nothing more than just changing the beat once, or reasonably cycle through beat modes.
- Pressing **up** increases the volume by 5 points; **down** decreases by 5 points.
  - Don't allow it to exceed the limits (above).
  - The user should be able to reliably press and release the joystick and have it change the volume just once. Or, the user should be able to press and hold the joystick and have it keep changing. No precise behaviour is required, just easy to control.
- Pressing **right** increases the tempo by 5 BPM, **left** decreases by 5 BPM.
  - Same requirements as the volume.

3.2.2 Accelerometer Requirements

- Allow the user to air-drum with the BeagleBone to play audio.
- Detect significant accelerations in each of the three axis (X: left/right, Y: away/towards, Z: up/down) and have each play three different sounds, one for each axis.
- For example, when the user “drums” the BeagleBone vertically (Z), have it play a base drum. For each other axis, use a different sound.
- It must be reasonably possible for a user to get just one play-back of sound per “air-drumming”. Therefore debouncing is likely required.
  - If the user shakes the board quickly, however, its OK to playback multiple occurrences of the sound.
  - User should be able to air-drum at least 120BPM without issue. (i.e., you cannot use huge debounce times).

3.2.3 Hints

- Make at least one separate C module (or C++ class) to handle the Zen-cape input. May be better to have multiple modules.
- On a separate thread, continually read the state of the joystick and accelerometer.
  - A reasonable start is to poll these inputs around every 10 ms (100 Hz). This should be fast enough to capture user inputs (such as accelerometer values).
- You'll want to debounce all joystick and accelerometer actions:
  - For example: If an action has be triggered for joystick up, then don't allow it to trigger another action for some time (say 100ms).
  - Do the same for each direction on the joystick, and each axis on the accelerometer. You may need different debounce “timers” for each action.
  - Think through how you can avoid copy-and-pasting large amounts of code 8 times!

- The accelerometer is an MMA8452Q by Freescale Semiconductor.
  - The part is connected to hardware I2C bus `I2C_1` at address 0x1C.
  - See the part's datasheet on the course website for details, such as:
    - Chapter 6 describes the registers the device exposes. I recommend looking at: `CTRL_REG1, STATUS, OUT_X_MSB, OUT_X_LSB` (and the same for Y and Z)
    - Note device must first be changed to the `Active` mode before it returns valid data.
    - **Reading a single register at a time seems to always return 0xFF** (cause unknown). So, read all the bytes in one operation (see next point). Also, **the first byte read during any read operation seems to be all 0xFF, so don't trust the first byte.**
    - If you read more than one bytes in a single read action, the device will automatically step through the registers. For example, reading 7 bytes starting at address 0x00 will return data for registers 0x00 through 0x06 inclusive. Hence it is not necessary to perform 7 different one byte reads.
    - I recommend not using any of the part's filtering/debouncing options, as it is simpler to get the hardware working without it. Plus it is easier to debug software than hardware settings. However, you are welcome to use any of the features it provides.
  - The part returns accelerations in terms of G forces. And since there's already 1G pulling down all the time, you may want to use different a threshold for the Z axis.
  - Given an array of bytes named `buff[]`, and the index of X's MSB and LSB (where x is a 16 bit value), you can create a 16-bit integer of those values using:
    ```
    int16_t x = (buff[REG_XMSB] << 8) | (buff[REG_XLSB]);
    ```
    - Note that the 4 lsb of the above value will be 0's because the device left-aligns its 12 bits of accurate data into the 16 bit value.

3.3 UDP Interface

Create a UDP interface which allows control of the beat box application. You'll use this interface in your NodeJS server (next section). I am not specifying what your interface should be; you get to design it any way you like. You may use the sample solution for As2 as a base. It must support:

- Changing the drum-beat mode directly (i.e., jumping from a standard rock beat to no beat).
- Changing the volume.
- Changing the tempo.
- Playing any one of the sounds your drum-beats use.

See the next section's requirements when designing your interface.

3.4 Memory Testing

We will run Valgrind on your code to look for incorrect memory accesses. Since there is no prescribed way to close the application, we are not going to be looking for memory leaks. If using Valgrind, you likely want to ignore all “leaks” that seem to be coming from `libasound.so`. While Valgrind-ing, your application's audio may stutter; this is OK.

4. Node.js Web Interface

4.1 Client Side Requirements

1. Must have a clear, well laid out interface. Likely requiring floating of elements, such as floating a `div` for the status to the right. Other layouts possible, must be at least as complex as sample.
2. Allow the user to directly select what beat to generate (none, standard rock, and any others you support).
   - Display what the current mode is.
   - Must update within 1s whenever the mode changes (either due to the web page or via the Zen cape input).
3. Allow the user to change the volume between 0 and 100. Support at least +/- buttons to change by 5 volume points.
   - Display the current volume as either a number or a graphic.
   - Must update display within 1s of the volume changing (such as user changing volume with Zen cape).
4. Allow the user to change the tempo between 40 and 300. Support at least +/- buttons to change by 5 BPM.
   - Display the current BPM as either a number or a graphic.
   - Must update display within 1s of the tempo changing (such as user changing tempo with Zen cape).
5. Allow the user to directly trigger the playback of each of the sounds found in your drum-beats. For example, clicking a “Base Drum” button.
6. Display the device's uptime in hours, minutes, and seconds (found via `/proc/uptime`). Poll for this every ~1s.
7. Display errors:
   - Create a box to display errors.
   - You must display meaningful error messages for at least the following errors:
     - NodeJS server is no longer running on the target (i.e., NodeJS server does not reply to a web-browser command for 1s). This assumes you have loaded the web page already, and then after that the connection fails.
     - `beatbox` C/C++ application not running (i.e., commands being relayed from the NodeJS server to the application generate no reply).

   Hints for Error Messages
   - Hide this box initially when the page is loaded.
   - When the server detects any error, have it send the client an error message.
   - When the client receives the error message, put the text in the “error-text” element and show the error-box.
     - Automatically hide the error box after 10 seconds using a timeout.
     - When using timers, be careful to clear any unneeded timers least they remain active and unexpectedly show/hide boxes.

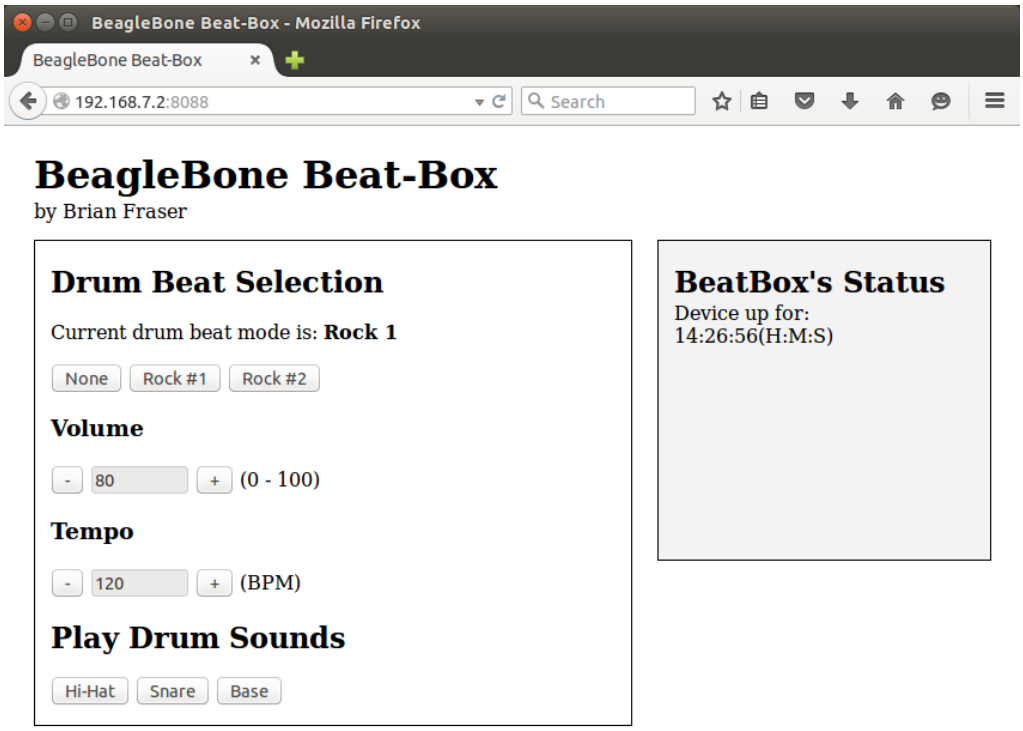

Figure 2: Sample screenshot of web page when it initially loads up.

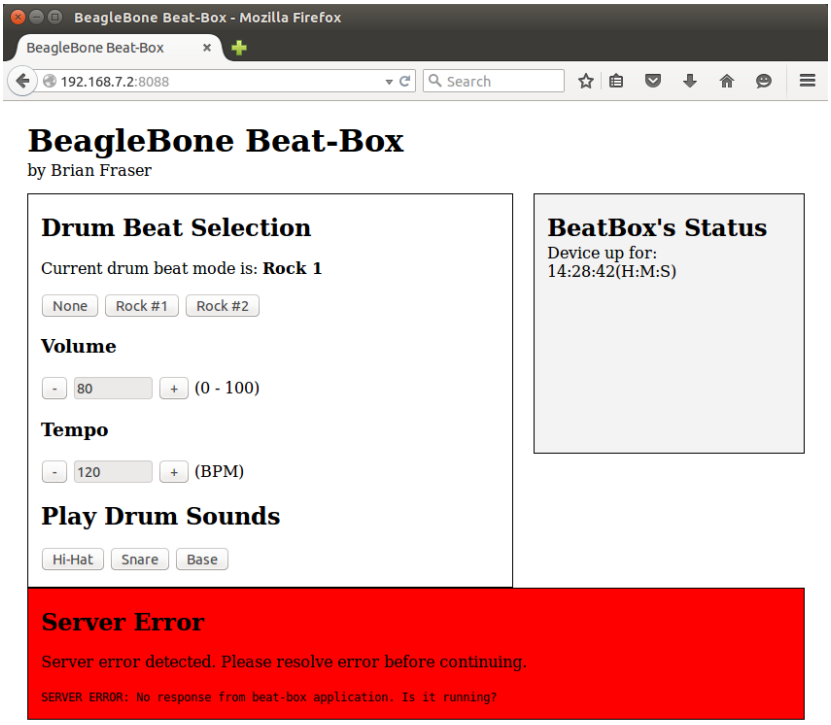

Figure 3: Sample image when an error is detected. Your error messages need not match this.

4.2 Server Side Requirements

1. Must be written using Node.js.
2. Support connections via HTTP on port 8088.
3. Relays commands between the client web browser and the C/C++ beat-box application.
4. Read device's current up-time (found via `/proc/uptime`).

4.3 Hints

- Change your UDP protocol as needed to make it easy to write the web interface.
  - For example, each command should generate some reply to indicate it was received, and so system can detect when the application is not running.
- For the error box:
  - Use a for the error box; give it an ID like “error-box”
  - Use CSS to hide the error box initially:
    ```
    #error-box { display: none;}
    ```
  - Show the error box in JavaScript code with:
    ```
    $('#error-box').show();
    ```
  - Hide it with:
    ```
    $('#error-box').hide();
    ```
- If using <input> elements to show the volume and tempo then make them read-only:
  ```
  <input type="text" id="volumeid" value="???" size="3" readonly/>
  ```

5. Deliverables

Submit the items listed below in a single ZIP file to CourSys: https://courses.cs.sfu.ca/

1. C/C++ code for the application (may be inside own directory)
2. NodeJS code (likely inside own directory)
3. Wave files for playback (likely inside own directory)
4. A `makefile` which builds both your C/C++ app and copies your wave files and NodeJS server to the public folder as specified in Section 2.

Since the assignment can be done individually or in pairs, if you are working individually you'll still need to create a group in CourSys to submit the assignment.

Remember that all submissions will automatically be compared for unexplainable similarities.