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 questions to the course discussion forum.
  - You may use any code you have written for this offering of CMPT 433. You may not resubmit code you or any one else has submitted for previous offerings.

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

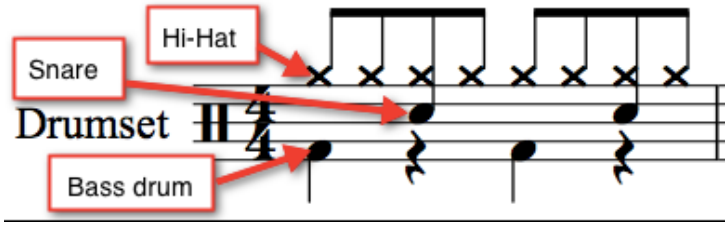

Figure 1: Musical score showing a rock beat.

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

Submit a single ZIP file containing your beat-box C/C++/Rust code, wave files, and NodeJS code. Must use CMake if building with C or C++. The build script must:

- Build your C/C++/Rust 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, 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 installed ALSA on the target and host, as described in the guide.
- We will have run `npm install` in the `~/cmpt433/public/myApps/beatbox-server-copy/` folder either from the target.

See course website for a sample `CMake` project.

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/C++/Rust program using the ALSA API[1], and play that audio through the Zen cape's head-phone output.
  - Audio playback must be smooth, consistent, and with low latency (low delay between asking to play a sound and the sound playing).
  - At times, multiple 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 (inclusive); 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 when needed (i.e., have functions that other modules can call to playback drum sounds when needed).
- Audio playback must be smooth, consistent, and with low latency (low delay between asking to play a sound and the sound playing).

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

Optional 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`
- When you are *first* completing the low-level PCM audio mixing code, first try setting your `main()` to something like the following pseudo-code:

```
initialize audio mixer
while(true) {
   play the base drum sound
   sleep(1);
}
```

  - Remove this code once you have written the beat-generation module/thread.

- After you can play one sound reliably, try using the same loop as above, but this time play 2 sounds at once (i.e, play base drum and hi-hat before `sleep(1)`).
- Beyond the low level audio mixer module, you'll likely want a higher level module which generates the drum beats, and allows other modules to request a sound be played.
  - You'll likely need a thread in here to continuously generating the beat.
  - Have your thread's sleep duration depend on the current tempo (beats per minute)

3.2 Required 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.
  - If the user presses and holds the joystick in, you may make it either do nothing more than just changing the beat mode once, or have it 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. And the user should be able to press and hold the joystick and have it keep changing. No precise timing 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. For this, when the user moves the BBG and Zen cape in direction, it will trigger a sound.
- Detect significant accelerations in each of the three axis (X: left/right, Y: away/towards, Z: up/down) and have each play a different sound, one for each axis.
  - You may assume the board is always held parallel to the ground (not on its side or upside down).
  - See this video for an explanation: https://www.youtube.com/watch?v=JmDie3wQgfk
  - The sound generated in response to the accelerations is in addition to any sound generated by the drumbeat modes.
- 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 large debounce times).

3.2.3 Hints

- Make at least one separate C module (or C++ class, Rust module/crate, ...) to handle the Zen-cape input. May be better to have multiple modules.
  - You may reuse modules you wrote for previous assignments during this offering of CMPT 433.
  - Your app must configure your board, as necessary, such that it runs fine after the BBG is rebooted. This may involve configuring GPIO pins or I2C commands.
    - If needed, you may need to wait ~330ms after exporting a GPIO pin before using it.
- 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 to 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!

- If you have a **Zen Green** (V1.0), then 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, 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.

- If you have a **Zen Red** (V1.1), then the accelerometer is an LIS331DLH by ST Microelectronics
  - The part is connected to hardware I2C bus `I2C_1` at address 0x18.
  - See the part's datasheet on the course website for details, such as:
    - Chapter 7 describes the registers the device exposes. I recommend looking at: `WHO_AM_I`, `CTRL_REG1`, `OUT_X_L`, `OUT_X_H` (and the same for Y and Z)
    - First, enable the chip using `CTRL_REG1`. You'll want to set it to power mode 1, and enable Z, Y, and X.
    - Next, I suggest you read the chip and see if you can get the expected value back on the `WHO_AM_I` register.
    - Finally, notice that the X, Y, and Z (high and low byte) registers are all in a row. You can do a multi-byte I2C read, starting at the lowest address number, to read all the registers in one operation. To make this work, you must add 0x80 to the register address of your read. Doing so reads the same address as otherwise but enables the auto-increment feature on the device's I2C address.
    - 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.

- General accelerometer hints:
  - First get the part working using command-line I2C tools. Then write your C program to read/write the I2C registers.
  - The accelerometers 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 Text Display

Once per second, print to the console the following:

- Beat mode (its number), format such as "`M0`"
- Tempo, format such as "`90bpm`"
- Volume, format such as "`vol:80`"
- Time between refilling audio playback buffer
  - Each time your code finishes filling the playback buffer, mark the interval/event.
  - Format: `Audio[{min}, {max}] avg {avg}/{num-samples}`
- Time between samples of the accelerometer
  - Each time your code reads the accelerometer, mark an interval/event.
  - Format: `Accel[{min}, {max}] avg {avg}/{num-samples}`

Sample output:

```
M0 90bpm vol:80  Audio[16.283, 16.942] avg 16.667/61  Accel[12.276, 13.965] avg 12.998/77
```

You may use the provided `intervaltimer.h/.c` code on the course website to mark each interval/event (record the timestamp), and to get the statistics based on these timestamps. You'll need to update/add your own enum to the .h file for the different periods you want to track.

You may output additional text to the terminal in response to events like changing modes/volume, or detecting an acceleration triggering a drum sound.

3.4 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 not use my sample assignment solution, but you may use any **example code**, or your assignment solutions 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.
- Shutting down the program gracefully

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

3.5 Memory Testing

We will run Valgrind on your code to look for incorrect memory accesses, and it must free all allocated memory (none lost, none still reachable).

You can ignore all "leaks" that seem to be coming from `libasound.so`.

While Valgrind-ing, your application's audio may stutter terribly and print errors from the `snd_pcm_writei()` call ("`AudioMixer: writei() returned..`"). These may be ignored

4. Node.js Web Interface

4.1 Upgrade Node Version

Execute the following commands on **both the host and target** to upgrade node to the latest stable version (using the tool named 'n'). This ensures all systems are running the same version.

```
sudo apt update
sudo apt install npm              # Installs ~500MB on BBG
sudo npm cache clean -f
sudo npm install -g n
sudo n stable                     # Updates Node.js to v20.11 (Feb 2024)
```

Then exit your shell and start a new one in order for it to call this updated version of Node.js.

4.2 Website Requirements

1. Must have a clear, well laid out interface. You'll likely need floating of elements, such as floating a `div` for the status to the right. Other layouts possible, must be at least as "nice" (or complex) as sample.
2. Allow the user to directly select what beat to generate (none, standard rock, etc.).
   - 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; support at least +/- buttons to change volume by 5.
   - 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; 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. Allow the user to terminate the C/C++/Rust program (must gracefully shutdown).
7. Display the device's uptime in hours, minutes, and seconds (found via `/proc/uptime`). Update this every ~1s.
8. 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 within 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 (a `div` likely) 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.
  - When the error has cleared, automatically hide the error box within at most 10 seconds.
  - If using timers, be careful to clear any unneeded timers least they remain active and unexpectedly show/hide boxes.

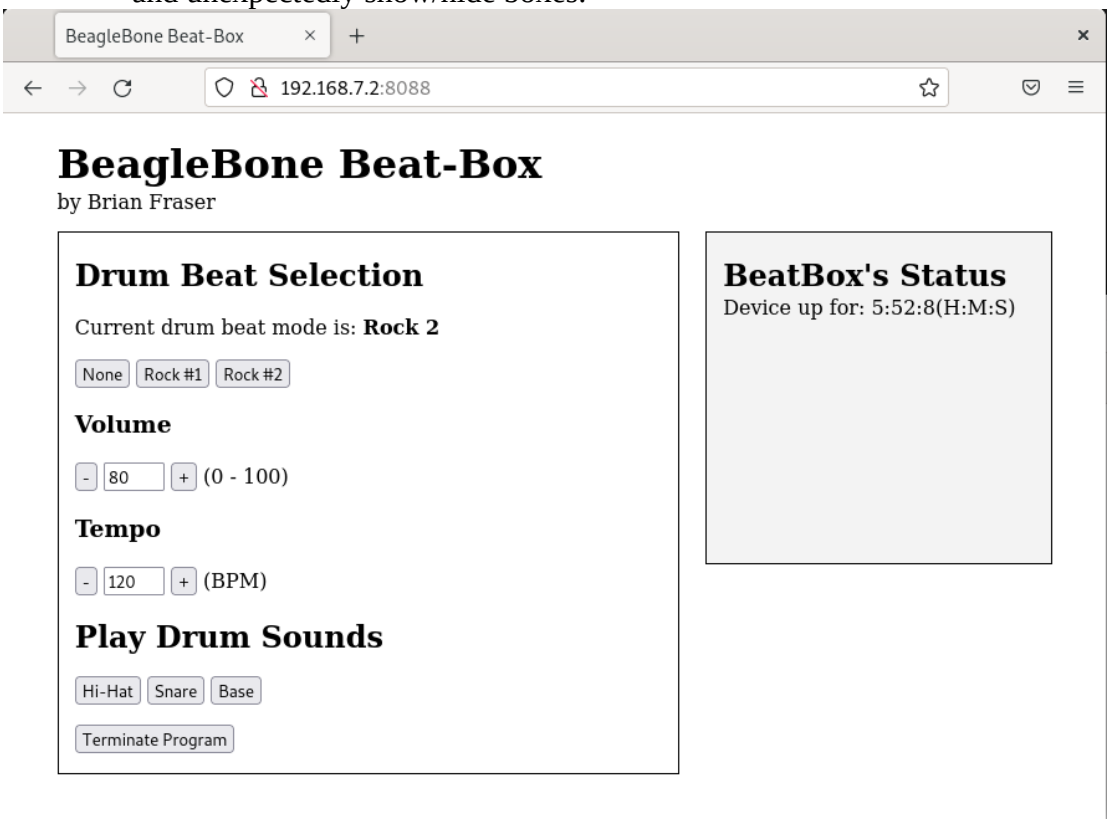

Figure 2: Sample screenshot of web page when it initially loads up.

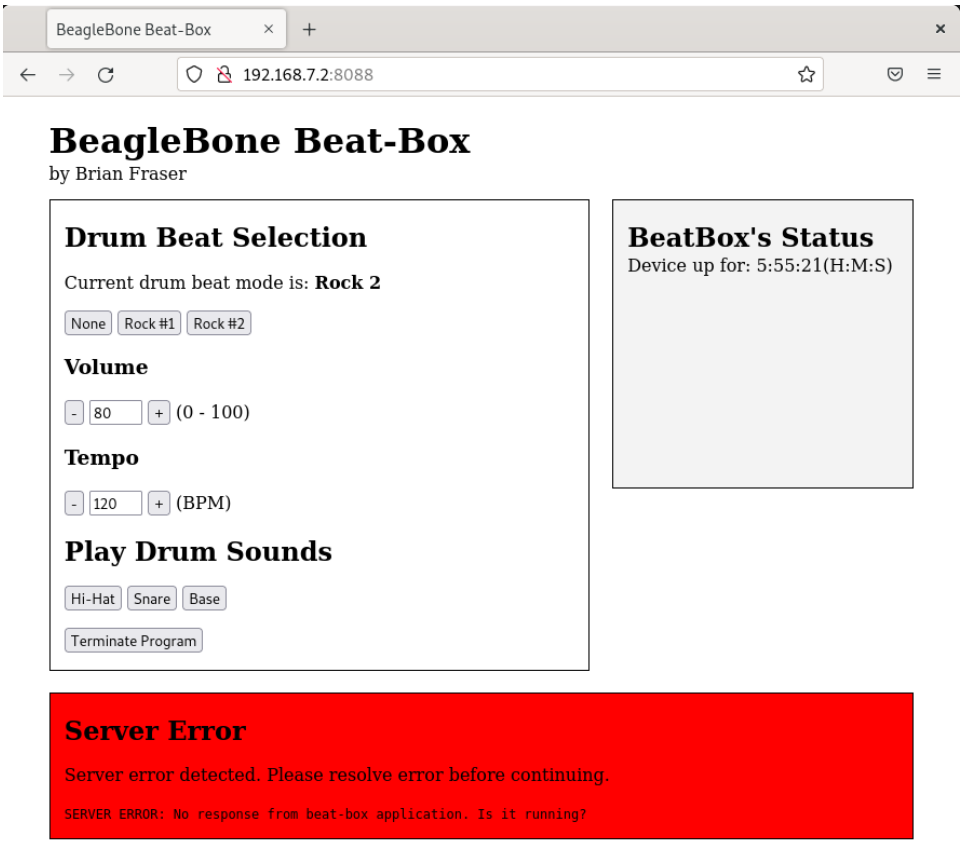

Figure 3: Sample image when an error is detected. Your error message need not match this.

4.3 Website Server Side Requirements

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

4.4 Optional 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 `<div>` 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://coursys.sfu.ca/

1. `as3-beatbox.tar.gz`
   Compressed copy of source code and build script (`Makefile`).

   Archive must include all necessary files to build your application, deploy the Node.js server, and the wave files. Hint: Compress the `as3/` directory with the command
   ```
   $ tar cvzf as3-beatbox.tar.gz as3
   ```

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 from both this semester, and previous semesters!

5.1 Informal Milestones

- How to start
  - If you want to work with a partner, start looking for one!
  - Follow the guides to get started:
    - Audio guide to be able to play sound
    - Setup your CMake project (for C/C++) to link with ASLA library
    - Work through getting the accelerometer working. No guide is provided for the device, so you'll need to work directly with the datasheet.
  - Read all sections of the assignment. Design HAL modules. Think about what modules your application will have.
  - Think about the application design. How will you generate a rock beat? How will you handle playing additional sounds in response to the accelerometer or UDP message?
  - Think about how you will shutdown the application correctly.
  - Get the low-level audio mixing routines working smoothly. Start with a simple `main()` playing the same sound each second to ensure it works smoothly.
- **Half done**
  - Low level audio playback working reliably.
  - High-level module plays a drum beat.
  - Perhaps also have a C program that can read the accelerometer.
- **Final checks**
  - Review the learning objectives for this assignment (see webpage).
  - Ensure your Node.js webpage can control the app. Ensure that changes in state due to the joystick (such as volume up) show up automatically on the web page within 1s.
  - Double check your final ZIP file for correctness! No last minute refactoring bugs!