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 or your partner have written for this offering of ENSC 351.
- Submit deliverables to CourSys: https://coursys.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 notes are often played on half-beat increments (called eight notes).

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

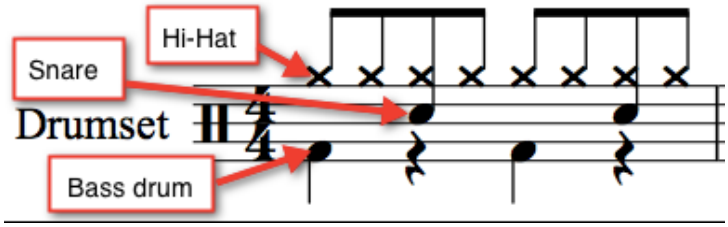

Figure 1: Musical score showing a rock beat.

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++ code, and wave files. When the TA extracts this ZIP file it must have a single Makefile which builds and deploys the application, and wave files in one command: `make`

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/`

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:

- All GPIO pins are exported, configured for GPIO, and set to output.
- We have a folder `$(HOME)/cmpt433/public/asound_lib_BBB/` containing the `libasound.so` library from the BeagleBone.:

You may find the following Makefile to be useful:

```
# Makefile for building embedded application.
# by Brian Fraser

# Edit this file to compile extra C files into their own programs.
TARGET= beatbox
SOURCES= main.c audioMixer.c

PUBDIR = $(HOME)/cmpt433/public/myApps
OUTDIR = $(PUBDIR)
CROSS_TOOL = arm-linux-gnueabihf-
CC_CPP = $(CROSS_TOOL)g++
CC_C = $(CROSS_TOOL)gcc

CFLAGS = -g -std=c99 -D _POSIX_C_SOURCE=200809L -Werror -Wshadow -Wall

# Asound Library
# - See the AudioGuide for steps to copy library from target to host.
LFLAGS = -L$(HOME)/cmpt433/public/asound_lib_BBB

# -pg for supporting gprof profiling.
#CFLAGS += -pg


all: beatbox wav

beatbox:
	$(CC_C) $(CFLAGS) $(SOURCES) -o $(OUTDIR)/$(TARGET)  $(LFLAGS) -lpthread -lasound

clean:
	rm -f $(OUTDIR)/$(TARGET)

# Copy the sound files to public directory.
wav:
	mkdir -p $(PUBDIR)/beatbox-wav-files/
	cp -R beatbox-wav-files/* $(PUBDIR)/beatbox-wav-files/
```


3. Wiring

The following is the required hardware configuration for completing this assignment.

- **Joystick** (see A2D Guide)
  - P9.32 (`VDD_ADC`) to +V to the joystick
  - P9.34 (`GNDA_ADC`) to -V to the joystick
  - P9.37 (`AIN2`) to X output on the joystick
  - P9.38 (`AIN3`) to Y output on the joysticks
  - Place joystick with breakout board's text ("Analog Mini Thumbsick") to the left, and its "`+ Y X -`" pins at the top and bottom.

- **8x8 LED Matrix** (see I2C Guide)
  - P9.01 (`Ground`) to "-" pin on the LED Matrix
  - P9.03 (`3.3V`) to "+" pin on the LED Matrix
  - P9.17 (`I2C1-SCL`) to "C" pin on the LED Matrix
  - P9.18 (`I2C1-SDA`) to "D" pin on the LED Matrix
  - Orient display with pins at the bottom, and display pointing out the top of the breadboard. (This is 180 degree rotation: text on the "backpack" board upside down).

- **GPIO Buttons**
  - All buttons:
    - Insert button across the trench on the breadboard.
    - P9.03 (`3.3V`) Connect left pin to 3.3V
    - P9.01 (`Ground`) Connect right pin through a 1K ohm resistor to ground
  - Connect the GPIO pin to read the button onto the right pin of button
    (Circuit: 3.3V ---- Button ---- [GPIO probe point] ---- Resistor ---- GND
    - P8.15 (GPIO) Mode (gray?) button's GPIO
    - P8.16 (GPIO) Base drum (red?) button's GPIO
    - P8.17 (GPIO) Snare drum (yellow?) button's GPIO
    - P8.18 (GPIO) Hihat (green?) button's GPIO

Use the provided hardware verification app to ensure that your hardware wiring matches the required setup. This wiring setup will be used by the TAs for marking, so your wiring should match it 100% (except possibly the colour of the buttons).

4. Beat-Box

You will create a beat-box application which can play different drum-beats on the BeagleBone using the USB-Audio Adapter included in your hardware kit.

4.1 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 USB Audio Adapter's headphone output
- You must have at least two threads:
  1. A low-level audio mixing thread which provides raw PCM data to the ALSA functions. This thread is in the provided audio code (which you must complete, and may adapt to your needs).
  2. Higher-level beat-generation thread which generates a rock beat and tells on your lower-level audio playback module (thread) to play rock beat sounds as needed.
- 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. The provided 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 sections 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). See Custom Push-Buttons; Section 4.3
- 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 wave values to generate the sound.

It is likely that the assignment will be updated to require some attention to some real time issues, such as thread priorities. This is likely to relate to the audio mixing thread in the provided code, and to the beat-generation thread describe here. It will likely add more code to the pthread_create() function call.

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

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 and hihat 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 is played.
  - You'll *need* a thread in here for continuously generating the beat.
  - Have your thread's sleep duration depend on the current tempo (beats per minute)

4.2 Quit

When the user types Q <enter> on the keyboard, exit the application.

The following C code may be useful to place in the middle of your `main()` (you may copy it without citing). Note that `getchar()` will wait until the user enters a character (and presses ENTER). It is blocking.

```
printf("Enter 'Q' to quit.\n");
while (true) {
      // Quit?
      if (toupper(getchar()) == 'Q') {
            break;
      }
}
```

4.3 Custom Push-Buttons

Wire up 4 custom push buttons with the following functionality. Colours are just suggestions; only the wiring matters. See the above Section 3. for Wiring.

When a button is pressed, begin the action stated below. There is no action for holding a button (i.e., make the button edge triggered on the pressing-edge of button; nothing for held, nothing for released).

Buttons

- Mode (gray)
  - On each button press it should then cycle to the beat (modes).
  - Default is the standard rock beat.
  - Order of cycle is: standard rock beat, custom beat(s), none (then repeat).
- Base Drum (red)
  - Play the base drum sound once, right now (or other sound, if using custom sounds).
- Snare Drum (yellow)
  - Play the snare drum sound once, right now (or other sound, if using custom sounds).
- Hi-hat (green)
  - Play the hi-hat sound once, right now (or other sound, if using custom sounds).

4.4 Joystick

The user should be able to reliably press and release the joystick and have it change the volume or tempo just once. And the user should be able to press and hold the joystick and have it keep changing slow enough to control it. No precise timing is required, just easy to control.

- Pressing **up** increases the volume by 5 points; **down** decreases by 5 points.
  - Don't allow it to exceed the limits (above).
- Pressing **right** increases the tempo by 5 BPM, **left** decreases by 5 BPM.
  - Same requirements as the volume.

4.5 8x8 LED Matrix

Use the 8x8 LED matrix to give the user feedback about what they are doing.

- **Drum Beat Mode**
  - Show what drumbeat mode we are in. This is what is displayed by default.
  - Display “M0” (none), “M1” (rock), or “M2” (custom); additional “M3”... if more.
- **Volume**
  - When the user is pressing, or has recently pressed up or down on the joystick, display the 2 digit number for the volume.
- **Tempo**
  - When the user is pressing, or has recently pressed left or right on the joystick, display the 2 digit number for the tempo (beats per second).
- **General**
  - Display 99 if the value is >100
  - Continue displaying the volume or tempo message for between about 0.5 to 1.5s after the user release the joystick. Then return to displaying the mode.

4.6 Text Display

TBA. Likely requiring you to display some combination of information about:

- Beat mode
- Volume
- Tempo
- Time between generating 8th notes
- Time between refilling audio playback buffer

4.7 Memory Testing

We will run Valgrind on your code to look for incorrect memory accesses and leaks. While Valgrind-ing, your application's audio may stutter terribly and print buffer underflow errors (running out of data); this is OK.

You may ignore all “leaks” or memory access that seem to be coming from `libasound.so` and are not related to your code. For example, you may ignore:

```
==1804== Syscall param semctl(IPC_GETALL, arg.array) points to unaddressable byte(s)
==1804==    at 0x49477E6: __libc_do_syscall (libc-do-syscall.S:47)
==1804==    by 0x49CEE6F: semctl_syscall (semctl.c:48)
==1804==    by 0x49CEE6F: semctl@@GLIBC_2.4 (semctl.c:94)
==1804==    by 0x48E9B1D: ??? (in /usr/lib/arm-linux-gnueabihf/libasound.so.2.0.0)
==1804==  Address 0xbd816000 is not stack'd, malloc'd or (recently) free'd
```

4.8 Hints

- You may reuse code from your Assignment 1 (from ENSC 351, this semester) which either you or your partner wrote.
- Make C modules for each input and output device. Give them easy to use .h files and test each module before integrating it with the rest of your application.
- You may assume that all GPIO pins are already exported, and that they are configured to be GPIO pins as input.
- On a separate thread (a module!), continually read the state of the joystick and buttons.
  - A reasonable start is to poll these inputs around every 10 ms (~100 Hz). This should be fast enough to capture user inputs (such button presses or joystick inputs).
- Don’t Repeat Yourself
  - Think through how you can avoid copy-and-pasting code numerous times.
  - For example, try writing one piece of code which can be called multiple times for controlling the joystick’s repeating behaviour (able to press once and change once; able to press and hold to have it keep incrementing slow enough to control it).

5. Deliverables

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

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

  Archive must expand into the following (without additional nested folders to find the `Makefile`)

  ```
  <assignment directory name>
   |-- Makefile
   |-- C/C++ code for the application (may be inside own directory)
   \-- Wave files for playback (likely inside own directory)
  ```

  `Makefile` must support both the `make` and `make all` commands to build your program to `$(HOME)/cmpt433/public/myApps/` plus it must copy your wave files to the public folder as specified in Section 2.
  (Do not use relative paths for getting to the `cmpt433/public/myApps/` directory because the TA may build from a different directory than you.)

  Hint: Compress the `as3/` directory with the command
  ```
  $ tar cvzf as3-beatbox.tar.gz as3
  ```

  You may use a different build system than `make` (such as CMake). If you do, include a file named `README` which describes the commands the TA must execute to install the necessary build system under Debian 11, and the commands needed to build and deploy your project to the `~/cmpt433/public/myApps/` directory. The process must be straightforward and not much more time consuming than running `make`.

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!

