ECS7012 Music and Audio Programming • Spring 2021

Assignment 2: Drum Machine

Handed out: Wednesday, 3 March 2021
Report and Code Due: 10:00, Friday, 26 March 2021

Introduction:

Your task is to create a digital audio system which plays sequenced drum loops with various tempos and styles, depending on input from several sensors.

Your drum machine will play drum samples which will be given to you in preloaded buffers. Each buffer contains a single drum sound. You will also be given a collection of patterns describing how the drum samples should be ordered into loops. Your task will be to create the audio code that plays the samples in the specified loop.

The speed of the loops and which loop to play at any given time will be selectable by sensors attached to the Bela board, including an accelerometer which measures the tilt of the board, a potentiometer and two buttons. Your code will handle reading the sensor data and using it to control the parameters of the audio sequencer.

Required Materials:
- Lab Kit (Bela Starter Kit and cables)
- 1 x 3-axis accelerometer board
- 1 x potentiometer
- 2 x buttons
- 1 x LED
- Resistors (10k for buttons, 470Ω for LED)
- Code template from QMplus

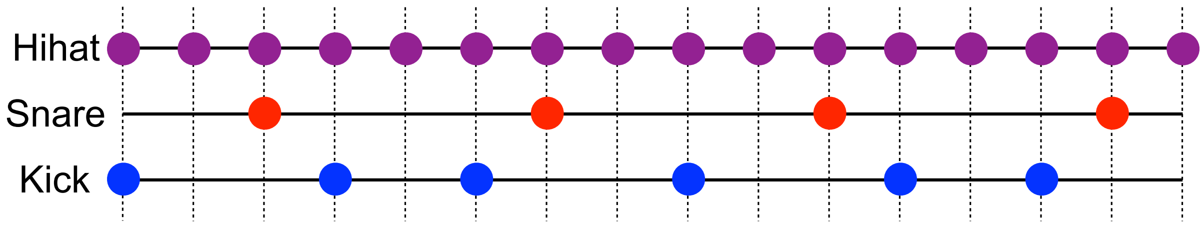
System Description:
You need to build a system with the following requirements. You can choose to follow the steps in the next section or build using your own design. In either case you must document your design decisions.

- Play multiple drum samples simultaneously, up to 4 drum samples
- Play samples back according to a defined looping pattern of 16 events (beats)
- Store and recall 5 different drum patterns
- Start and stop playback with button press
- LED that flashes on each beat, like a metronome
- Change playback tempo with potentiometer
- Select drum pattern with orientation of accelerometer
- Turning the accelerometer upside down plays the drum samples of the current drum pattern backwards
- BONUS: detect when accelerometer tapped and play drum fill pattern once

Instructions:

1. Overview of the task

Your task in this assignment is to create an audio *sequencer* system which is capable of playing sampled drum sounds at precise times. The sounds themselves are held in buffers (each buffer containing one single drum sound), and the patterns are specified in arrays. A simple example pattern is shown below, which would play in a loop of 16 events:



Your job will be to play the sounds at the right time, to manage the overall timing of the sequence (keeping track of where you are in the sequence and when the next event should arrive), and to select between different patterns depending on sensor input data.

2. Playing a drum from a sample buffer

In this step, you should write code so that pressing a button plays a drum sound.

See the appendix for how to wire the button to the Bela. Your button-reading and audio code should go in `render.cpp`. The drum sample buffers are stored in the array `gDrumSampleBuffers` (where each element of the array is a buffer holding a different drum sound). The lengths of each sample are stored in the array `gDrumSampleBufferLengths`. For example, `gDrumSampleBufferLengths[0]` holds the length of the drum sample stored in `gDrumSampleBuffers[0]`.

To work with the button:

- Initialise the GPIO input on the correct pin (in `setup()`)
- Read the button value and check if it has changed from unpressed to pressed (in `render()`). You will need to save the last value in a global variable. *Hint: see the Week 4a example where we looked at how to tell if the button was just now pressed.*

To trigger the sound, check if the button was just now pressed, using the code you just wrote. When this happens, you should start playing sound from the buffer (and reset the variable). A variable `gReadPointer` has been declared for you, which you can use to play sound from the drum buffer. You need to do three things:

- Start the read pointer at 0 when the button is pressed
- Copy samples from the drum buffer to the output audio buffer in `render()` (remember both channels!), incrementing the read pointer after each sample.
- Stop when the read pointer reaches the end of the drum buffer.

Hint: think about what the default value of gReadPointer should be before you press the button. Implement this in the top of the file.

3. Multiple drums at once

Playing real drum patterns will often require multiple sounds to be active at once. Here, we will create a scheme for playing up to 16 sounds at once; these could be different drums or even multiple copies of the same drum (e.g. if playing the same sound again before the first one has finished).

In this step, rather than having one read pointer, we will have an array of them. First replace `gReadPointer` with an array called `gReadPointers`, which contains 16 elements. Then create a second array of 16 elements, type `int`, which will hold which buffer is associated with each read pointer. You might call this `gDrumBufferForReadPointer` or similar.

3a. Starting a drum sound

Fill in the function called `startPlayingDrum()`. This function should be called whenever you want to play a new drum sound. It takes as an argument the index of the drum to play (i.e. which buffer within `gDrumSampleBuffers`). Your code should do the following:

- Find the first read pointer that is not already being used to play a drum (*Hint: use a for() loop to go through and check each element of the array. The* `break;` *statement lets you stop the loop once you find a read pointer that isn't being used.*)
- Use `gDrumBufferForReadPointer` to indicate which buffer should be played, and reset the read pointer to 0 to make it start playing.
- If there are no read pointers free (all 16 in use; unlikely), you can return without playing the sound. (There are other options, such as voice stealing, but we don't need to do that here.)

3b. Playing the drum sounds

Once you have created these two arrays, your code in `render()` should go through each read pointer in the array (*Hint: for() loop*). Check whether the read pointer is actually associated with a drum sound (*Hint: use* `gDrumBufferForReadPointer`*; what kind of value might you assign to indicate that the particular read pointer isn't used right now?*). Also check whether the read pointer has reached the end of the buffer, and if so, change `gDrumBufferForReadPointer` to indicate that the pointer is no longer active.

For every active read pointer, mix the drum sound from the buffer into your audio output. *Hint: You may need to reduce the overall level of the output to avoid clipping with multiple drums.*

3c. Testing multiple drums

Attach two buttons to your BBB. Extend your code from Step 2 to play two different drum sounds. When one of the triggers is set, call `startPlayingDrum()` to initiate the sound. You should be able to play both at the same time without glitches or distortion. You should also be able to press the button quickly to re-trigger a drum without cutting off the previous sample.

4. Playing drums in a loop

In this step, you will play looping sequences of drum sounds based on pre-stored patterns. The patterns are stored in arrays; which pattern to play and how fast to play it will ultimately be selectable by sensors. *Note: you no longer need the buttons to trigger drum sounds as they did in step 3; that was only for testing.*

4a. Create a simple metronome

You have been given a function named `startNextEvent()`. You will use this function to trigger the next event in the pattern. For this step, add code to `startNextEvent()` so that it always plays the same drum sound every time it is called (*Hint: startPlayingDrum()*).

Next, add code to `render()` which counts the number of audio samples that have gone by (*Hint: you will need a global variable to save the number of samples; refer to the Week 7a exercises*). When it reaches the length of time specified in the global variable `gEventIntervalMilliseconds` (already given to you), call `startNextEvent()` and reset the counter. (*Hint: how do you convert from the interval in milliseconds to the number of samples needed?*)

When you have finished this step, you should hear a drum sound playing at a regular interval, like a metronome.

4b. Start and stop

Make one of the buttons start and stop the metronome. Read the button inside `render()` as before, and add code to set the global variable `gIsPlaying` to 1 or 0. Then, within your `render()` function, check the value of `gIsPlaying` to decide whether to trigger new events. *Note: when you stop the loop, the currently playing samples should finish playing, so don't use gIsPlaying to cut off all the audio! Just check it to decide whether or not to start further drum sounds.*

At the end of this step, you should be able to start and stop your metronome by pressing the button.

4c. Change tempo with the potentiometer

Wire up the potentiometer to one of the analog inputs on the cape (see appendix for circuit diagram). Inside `render()`, you should also add code to read the potentiometer value (`analogRead()`). Remember that there are two audio samples for every analog input sample (i.e. the sample rate is 22.05kHz compared to 44.1kHz for audio).

When you get the analog input, you need to map it to the tempo of the loop by changing the value of `gEventIntervalMilliseconds`. (Remember, smaller interval = faster tempo.) Make it so the full range of potentiometer values maps to an interval range of 50-1000ms. (*Hint: use the* `map()` *function.*)

When you finish this step, you should have a metronome whose tempo you can change by turning the potentiometer.

4d. Make the LED blink on each beat

In this step, you should make the LED blink at each beat. See the appendix for how to wire the LED. You will also need to add code to `setup()` to initialise the GPIO pin to be an output. In `render()` you will need the LED to stay lit for more than 1 sample so you can see it (*Hint: see Week 7a slides.*)

When you finish this step, the LED should blink with each successive beat.

4e. Play a drum pattern

You are given a two-dimensional array `gPatterns` which holds the sequence of drums to play. The first dimension of the array is which pattern to use; the second dimension is the index of the events within a pattern. For example, `gPatterns[0]` will hold the first drum pattern; `gPatterns[0][0]` would be the first event within the pattern, followed by `gPatterns[0][1]`, etc. The length of each pattern is stored in the array `gPatternLengths`.

In this step, you should change the code inside `startNextEvent()`. Instead of always playing the same drum sound, read the next event in the pattern to decide which drum(s) to play. You should do the following:

- You have been given two global variables: `gCurrentPattern` and `gCurrentIndexInPattern`. These hold which pattern to play and where within the pattern is currently being played. Each time `startNextEvent()` is called, look up the current event in `gPatterns` using these two indices. Figure out which drums this event contains and play them. You have been given a function `int eventContainsDrum(int event, int drum)`. This function returns 1 if the given event contains the given drum index (since an event may contain more than 1 drum). *Hint: use a for() loop to check for each drum; if eventContainsDrum() returns 1, then play that sound.*
- Increment the index within the current pattern. If you get to the end of the pattern (*Hint: check gPatternLengths*), reset the index to 0. Notice that your index within the pattern works very much like a circular buffer for audio samples.

At the end of this step, you should hear a complete drum pattern with multiple drum sounds, playing in a loop. The tempo of the loop will be controllable with the potentiometer. You're nearly there!

5. Select patterns with the accelerometer

You have been given a 3-axis accelerometer. This device creates 3 analog signals whose values are proportional to the acceleration in that axis. The X and Y axes are parallel to the breadboard. The Z axis is up/down assuming the breadboard is lying flat on a desk.

See the appendix for how to wire up the accelerometer. You will use `analogRead()` inside `render()` to read the values from each axis.

Important Note 1: the accelerometer is designed to measure acceleration *in either direction*, but the voltage it produces is always positive. Therefore, zero acceleration (i.e. sitting still or moving at constant velocity) will be somewhere roughly halfway between 0V and 3.3V. But the exact value will depend on the particular device and the resistors used, so you can't assume it is always 1.65V. Therefore, you may have to measure it experimentally and include these values in your code (or read values at startup, during the first few samples of `render()`, and use these as a reference).

Important Note 2: gravity is an acceleration! When the accelerometer is plugged into the breadboard and resting on the desk, you should measure **0g** (no acceleration) in the X and Y axes, and **1g** in the Z axis. Turn the board on its side (90°), and now you will measure +/-**1g** in either the X or Y axis, depending on which way you have turned it (and **0g** in the Z axis). Turn the board upside-down, and you will measure **-1g** in the Z axis. *Hint: the full range of output voltage on the accelerometer corresponds to -1.5g to 1.5g in each axis.*

5a. Detect board orientation

Based on the acceleration in each axis, you should be able to determine which way the board is facing. By default you will read around 1g in the Z axis and 0g in the others (*again, remember, 0g does not mean 0V!*). You need to design a system that recognises any of 5 orientations (note: depending on how your accelerometer is plugged into the board, the signs and axes may be different, but you will always have these 5 orientations):

- Resting flat (X = 0, Y = 0, Z = positive)
- Turned vertically on left side (X = negative, Y = 0, Z = 0)
- Turned vertically on right side (X = positive, Y = 0, Z = 0)
- Turned vertically on front side (X = 0, Y = negative, Z = 0)
- Turned vertically on back side (X = 0, Y = positive, Z = 0)

Write code within `render()` to read each axis and make a decision about what orientation the board is in. You don't necessarily need to check the accelerometer every single sample: reading it 100 times per second or so would be sufficient. Better yet would be to put a lowpass filter on each of the accelerometer signals to reduce noise. (*Hint: use a block of if() / else if() statements, but remember that the values in each axis may vary by a bit. You'll need some degree of tolerance so if, for example, X and Y are not* exactly *0g, you still detect the right orientation. Hysteresis will also help make the changes cleaner.*)

Initially, use `rt_printf()` statements so you can tell if your detection is working correctly. (*rt_printf() is like regular printf() but performs better in this real-time context. But if you print every single sample it will surely overload the system, so use sparingly!)*

5b. Change pattern depending on orientation

Based on the orientation information in the previous step, change the value of `gCurrentPattern` so that a different pattern plays in each orientation of the board. Remember that the patterns may be of different lengths! Each time you change `gCurrentPattern`, use modulo arithmetic to make sure `gCurrentPatternIndex` stays within the length of the new pattern (*Hint: check gPatternLengths*).

At the end of this step, your drum machine should play 5 different patterns depending on which way the board is turned.

5c. Play samples backwards when the board is upside down

There is one further orientation we have not explored: the board upside down will produce negative Z-axis acceleration and 0 in the X and Y axes. Add code to `render()` to detect this orientation. Don't change the current pattern in this case. Instead, set the global variable `gPlaysBackwards` (already defined for you). Then in your audio code, when this variable is set, you should make every drum sample play backwards instead of forwards. *Hint: make the read pointers count downward instead of upward. Now how do you check if you've reached the end of a drum sample?*

Now you should have a complete working drum machine! Tempo is adjustable by potentiometer, start/stop by the button, and selecting patterns by the orientation of the board. See below for some more features you can add if you're interested.

Bonus Step (up to 5% extra marks). Tap to add a fill

In the final step, we will temporarily play a different drum pattern (a fill) whenever we firmly tap the board. Drum fills are often used at the end of a section of several bars; they are usually played once rather than as a repeating pattern.

You have been given a variable `gShouldPlayFill`. You should set this in `render()` whenever you recognise a tap on the accelerometer. In the audio code (in `render()`), you should check the value of this variable. If it is set, you should start playing a new pattern. When you get to the end of this new pattern, you'll go back to what you were playing before, so you should save the value of `gCurrentPattern` inside the global variable `gPreviousPattern`. Then, set the value of `gCurrentPattern` to the special value `FILL_PATTERN` and reset `gCurrentPatternIndex` to 0. This will cause the fill to play next. *Hint: don't forget to reset gShouldPlayFill inside render().*

In `startNextEvent()`, when you get to the end of the current pattern, check whether the pattern was a fill (*Hint:* `if(gCurrentPattern == FILL_PATTERN)`). If so, set `gCurrentPattern` back to the value you stored in `gPreviousPattern`, which will cause the drum machine to go back to the pattern it was playing before.

Tapping the accelerometer will produce a temporary spike in acceleration. If you're tapping the surface of the breadboard, the spike should be primarily in the Z axis, but tapping in other directions is possible. For this step, you will need to design code to check for a spike in acceleration, and set `gShouldPlayFill` to 1 whenever you detect one. *Hint: try implementing a high-pass filter on the accelerometer data. This will pass the spikes while blocking the constant acceleration due to gravity. You can implement a filter on the accelerometer data exactly as you did on the audio data in Assignment 1: save the previous inputs and outputs and multiply by the correct filter coefficients. Remember that the sampling rate of the accelerometer, like all analog inputs on Bela, is half the audio rate (22.05kHz).*

Your submission should consist of the following:

1. Your **source code**. All .cpp and h files that you edit or create for the project including main.cpp and render.cpp.

- Don't forget to comment your code! Uncommented or illegible code will receive a reduced mark.
- Please submit a zip file containing your source code and your report via QMPlus.

2. A **3 to 4-page report (PDF format)**. Include the following information (though your report does not have to be structured in this way):
   - Describing your design process.
   - Details of your implementation.
   - State diagram explaining how your implementation works.
   - Cite your sources! Any code or designs found to come from another source without attribution will be treated as a case of plagiarism, and may be referred to the university for further action. Use the Harvard Style for citations.
3. Quick (2-3 min.) **video demo** of your drum machine in action. This can be recorded with a phone or laptop camera; production value is not important. It should show you starting and stopping the drum machine, changing the tempo with the potentiometer, and tilting the board in different directions to get different patterns (including upside down to play the samples backwards). If you have implemented the bonus step then please show that as well. If you have an external speaker that you can use with Bela, then use that to record the sound. If you don't have a speaker, you could try a large pair of headphones or holding the headphone directly up to the microphone. Sound quality is not especially important, but it should be possible to tell what the drum machine is playing.

Marking Criteria:

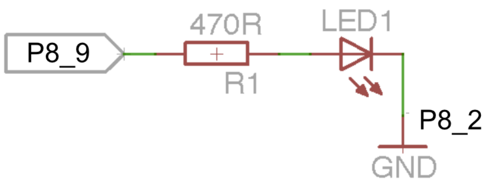
Theory	Demonstrates an understanding of the theory being implemented.	30%
Implementation	Code functions correctly, is well-commented, and design decisions are explained/justified in the report.	30%
Evaluation	Evidence demonstrating the implementation's success and discussion of any weaknesses.	30%
Report	Professional presentation of a technical report; sources are cited; figures are properly referenced and of sufficient resolution to be legible.	10%

Appendix: Circuits

Note: pin numbers in the schematics may be different in your implementation

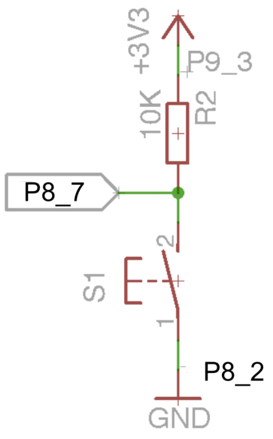
LED

Attaching an LED requires a resistor (range between 220Ω and 470Ω, depending on the brightness you want). Put the LED and resistor in series, making sure the short lead of the LED goes to ground. 2 wires to Bela required: one to the GPIO output pin, one to ground. *Note: your GPIO pin numbers could be different than the ones listed here; remember to refer to the pin diagram to tell you which pin corresponds to which GPIO number.*



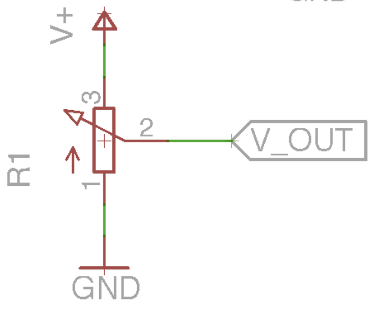
Button

The button requires a pull-up resistor (10kΩ suggested; brown-black-orange). The resistor goes between +3.3V and one terminal of the button. The other terminal of the button attaches to ground. **Never use 5V on Bela!** 3 wires to Bela required: one to +3.3V, one to the GPIO input pin, one to ground.



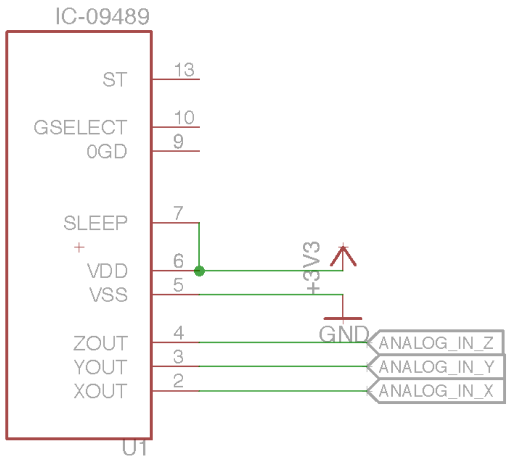
Potentiometer

The potentiometer is attached with 3 wires. One end of the potentiometer goes to **+3.3V** (which can be found on pin 3 of P9), the other end goes to ground. The **middle** terminal of the potentiometer goes to the analog input on the Bela cape.



Accelerometer

The accelerometer is powered by +3.3V and connects to three analog inputs on the cape. Five wires are required: 3 analog inputs, +3.3V and ground. In addition, you will need a wire between the +V input and the SLP pin to enable the accelerometer.


IC-09489
ST
GSELECT
0GD
SLEEP
VDD
VSS
ZOUT
YOUT
XOUT
13
10
9
7
6
5
4
3
2
+3V3
GND
ANALOG_IN_Z
ANALOG_IN_Y
ANALOG_IN_X
U1