Drum machine and sequencer for the BeagleBone Black and Bela

Iuliu Teodor Radu

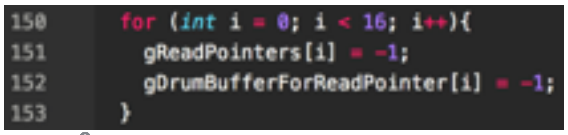
Introduction

The “drum machine” at hand is essentially an audio system that plays sequenced drum loops in several patterns and different tempos which are selectable by inputs from an accelerometer and a potentiometer. In more detail, features of this system include playing multiple drum samples simultaneously, playing back loops based on specified looping patterns of variable lengths, starting and stopping playback with a button, adjusting the tempo of beats with a potentiometer, metronomically flashing a LED on the main beats based on tempo, changing looping patterns based on the board’s orientation, and detecting a tap on the z-axis in order to carry out an intermediate looping pattern.

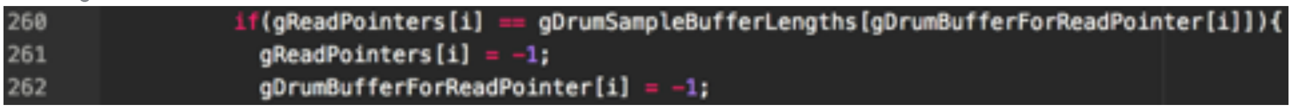
I will now walk through the implementation details of this system while describing the main challenges that I have been presented with during the design process.

Buffers, counters, and read pointers

In order to make sure that iteration through drum buffers happens correctly, I have first implemented a button which causes the attack of a specific drum. A challenge presented at this stage was deciding what to do with the read pointer for the drum buffer once the drum sound has played. After stopping the read pointer from incrementing to prevent reading past the end of the drum buffer array, I have decided to set the read pointer to -1, in order to show that it is not currently in use. By extension, I have decided to set all of the counters that are not being used to -1, and to increment them automatically if they are set to 0.



```
for (int i = 0; i < 16; i++){
  gReadPointers[i] = -1;
  gDrumBufferForReadPointer[i] = -1;
}
```



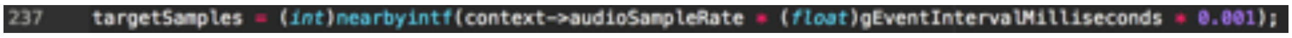
```
        if(gReadPointers[i] == gDrumSampleBufferLengths[gDrumBufferForReadPointer[i]]){
          gReadPointers[i] = -1;
          gDrumBufferForReadPointer[i] = -1;
```

Playing multiple sounds

At this stage, to be able to play multiple drum sounds, I have made an array with multiple read pointers so that I could iterate through drum sound buffers simultaneously. In order to use them, I have implemented the startPlayingDrum function, which looks for a read pointer that is not engaged i.e. a read pointer that is set to -1. I have gone on to test multiple drums sounding together by installing a second trigger button and linked a second drum to it. Within the render function, I have then added all of the active drums into the output buffer. I have decided not to limit the maximum volume of the output buffer, as the drums would have to be played in an unmusically quick succession for the output to clip. Also, even when multiple drums are mixed on the same beat, the output is unlikely to clip because of the differing envelopes that each instrument has. If this system were to be used in a situation where we want to add more channels and mix more instruments, then there would need to be a mixer that would mitigate this problem by balancing the volume.

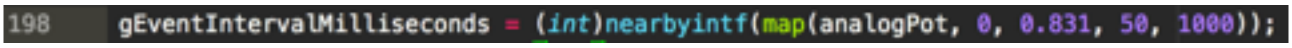
Timing and looping through patterns

I have implemented a metronome that would control the timing for the sequenced patterns that could be adjusted with a potentiometer. I converted the interval duration in milliseconds that would pass between each strike of the metronome into a sample interval as follows:



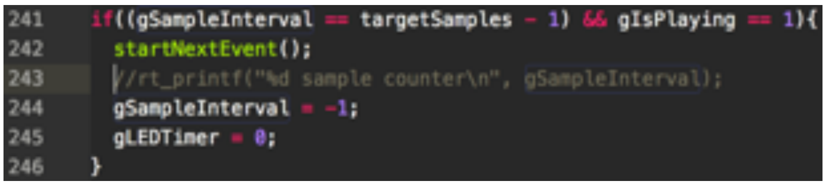
```
targetSamples = (int)nearbyintf(context->audioSampleRate * (float)gEventIntervalMilliseconds * 0.001);
```

This takes the time that should pass between beats in milliseconds, casts it to a float, multiplies it by the sample rate, 44100.0, multiplies by 0.001 to adjust the sample rate from seconds to milliseconds, rounds the entire result to the nearest integer, and casts all this to an integer. Attaching the potentiometer presented a difficulty. Since the analog input range is between 0V and 4.096V and the input voltage is 3.3V, I expected to have to map the potentiometer output values to between 0 and 0.806. However, as I was experimenting, I saw that I was getting a result all the way up to 0.831. Therefore I implemented the potentiometer to millisecond mapping as follows:



```
gEventIntervalMilliseconds = (int)nearbyintf(map(analogPot, 0, 0.831, 50, 1000));
```

After all this, I made a function that flashes a LED for each metronome beat and a function that detects the active or not state of the sequencer based on a button toggle.

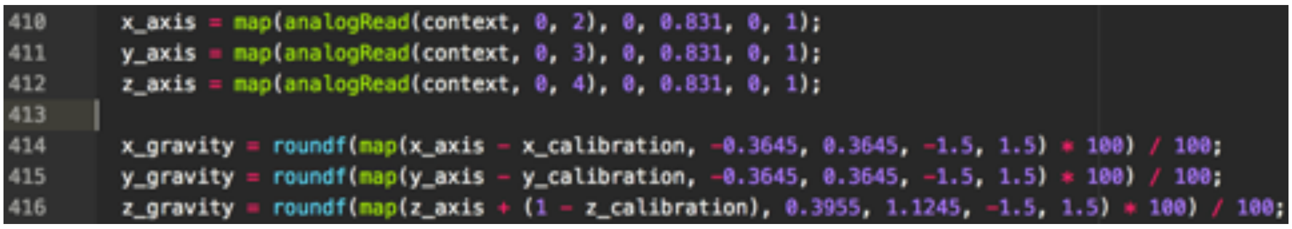


```
if((gSampleInterval == targetSamples - 1) && gIsPlaying == 1){
  startNextEvent();
  //rt_printf("%d sample counter\n", gSampleInterval);
  gSampleInterval = -1;
  gLEDTimer = 0;
}
```

Further on in the code, gSampleInterval is used to trigger the startNextEvent function. gSampleInterval is then set to -1 because before the end of the current for loop iteration it gets incremented. Therefore, on a subsequent iteration it would be 0 or positive. Following this, I have further developed the startNextEvent function to use the eventContainsDrum function in order to detect which drums are present on the current position of the pattern. Since eventContainsDrum can only detect individual drums due to the fact it needs to bit-shift an octet and bitwise-and it with the event, it needs to be run in a for loop the same amount of times as the number of drums.

Hooking up an accelerometer

I have decided to calibrate the accelerometer at the beginning of *render* by reading initial axis information into three variables. I have taken what I have learned from the potentiometer earlier and mapped the values of the analog input from between 0 and 0.831 to 0 and 1. Since I assumed that the axes read do not behave equally in relation to one another, I have used the calibration variables to offset any updated axis information in order to determine a gravity variable. Knowing that each axis on the accelerometer is able to detect a gravity between -1.5 and 1.5, I have mapped and fine tuned each axis by observing how off centred it is. The x and y axes were similar in terms of how off centre they were, while the z axis was a bit more extreme. Therefore, I came up with the following mapping for obtaining a clear gravity reading from each axis:



```
x_axis = map(analogRead(context, 0, 2), 0, 0.831, 0, 1);
y_axis = map(analogRead(context, 0, 3), 0, 0.831, 0, 1);
z_axis = map(analogRead(context, 0, 4), 0, 0.831, 0, 1);

x_gravity = roundf(map(x_axis - x_calibration, -0.3645, 0.3645, -1.5, 1.5) * 100) / 100;
y_gravity = roundf(map(y_axis - y_calibration, -0.3645, 0.3645, -1.5, 1.5) * 100) / 100;
z_gravity = roundf(map(z_axis + (1 - z_calibration), 0.3955, 1.1245, -1.5, 1.5) * 100) / 100;
```

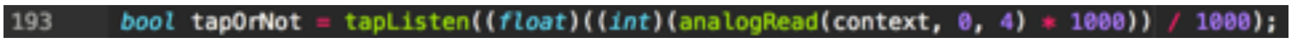
This mapping happens within a calculateAxes function, which gets called once every 300 analog frames. As can be seen in the above figure, initially at startup, the board should be placed on an even surface, in order to set the x_gravity and y_gravity to 0 (since the x_axis reading and x_calibration reading will be equal etc.) and the z_gravity to 1. I have rounded the final reading to two decimal places, as we do not need extreme accuracy when detecting the orientation of the board.

I have made the design decision to snap the gravity information for each axis into one of three states: -1 indicating negative gravity, 0 indicating an axis lying horizontally, or 1 indicating positive gravity. I have also implemented a Schmitt trigger, that does not let this value change unless there is a certain amount of difference between the new value and the old one. However, I saw that snapping the axes to one of three values was more than enough for getting a clean reading and not quickly toggling between states. This was due to a little bit of variation and overlapping between the final readings of each axis. So the Schmitt trigger is still there, except I have set its tolerance threshold to 0.0, so that it does not make any difference.

I have defined a setPattern function, that gets called once every 300 analog frames that uses the accelerometer values and sets the pattern accordingly. The sequencer plays each drum sample backwards if the board is upside down. I have made a conditional block by using gPlaysBackwards where the gReadPointers decrement after being set to the buffer size minus 1 in order to play backwards.

Tap to add a fill

I have implemented a second order Butterworth Filter applied to the raw data coming from the z-axis analog channel correct to 4 decimal places to detect spikes in its rate of change. I have done this by implementing a tap listener that runs at the beginning of each analog frame. The tapListen function returns true if a significant value is passed through the high pass filter. I considered that the values passed through have to be greater than or equal to 0.015 given a 10khz cutoff frequency of the filter.

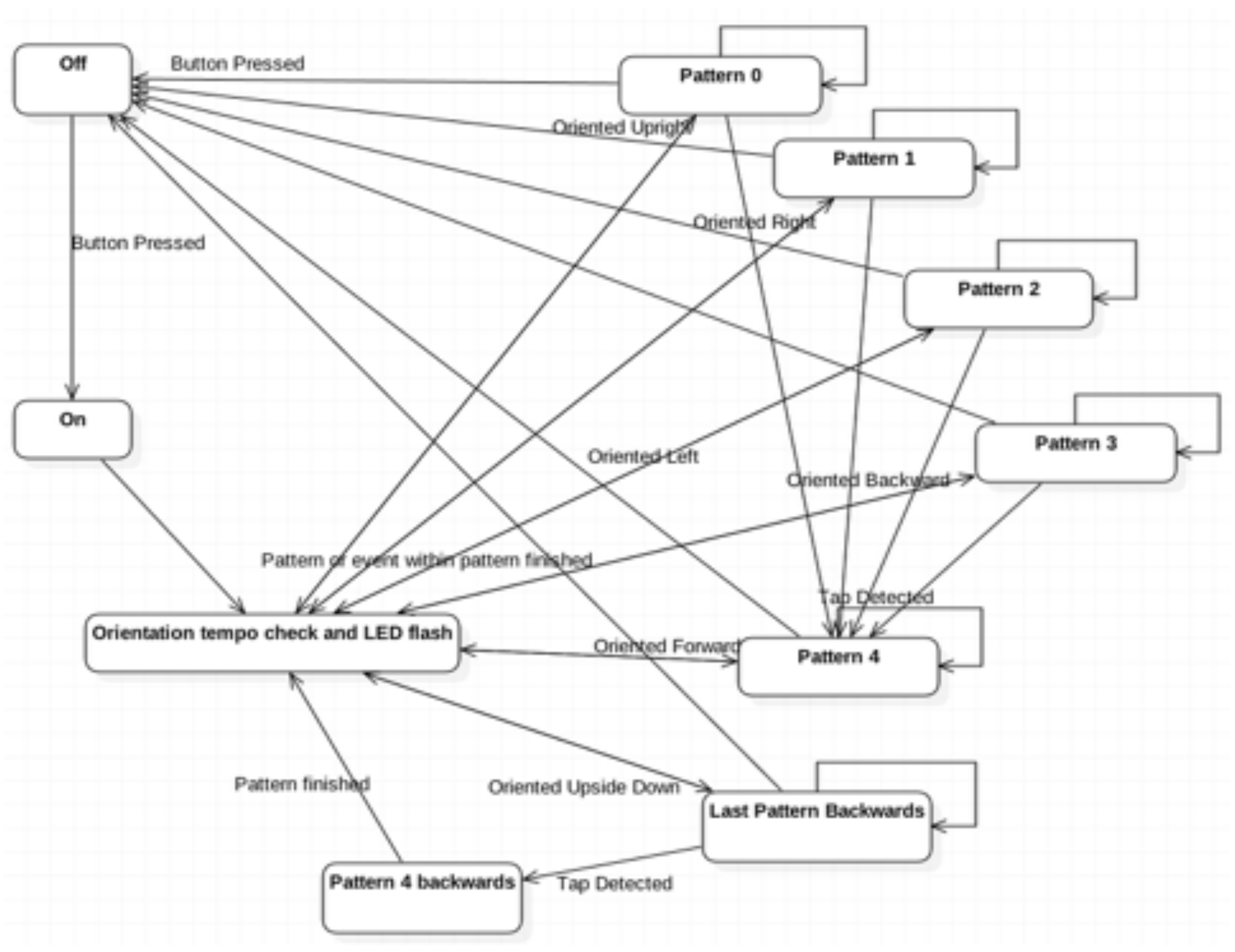


```
bool tapOrNot = tapListen((float)((int)(analogRead(context, 0, 4) * 1000)) / 1000);
```

Depending on the outcome of tapListen, I decided whether to set gShouldPlayFill to 1. Since tapListen would continue setting gShouldPlayFill for the period that the accelerometer's z-axis is vibrating, I considered it a wise choice to implement a debounce that would last for 1000 analog frames after each detected tap, preventing tapListen from triggering twice in a very short space of time.

State diagram

In the state diagram that follows, the illustrated pattern states could be subdivided into further states depending on which drums are sounding for each event. Therefore, I would like to warn that the "button pressed" action returning the machine to an off state can happen within each individual pattern. Indeed if a pattern has been stopped before it finished, when the on state is activated again, the pattern resumes from where it left off before checking the orientation of the board. For further clarification, both the tempo and orientation check can happen after each event within each pattern, except the tempo becomes effective immediately, while the orientation decides the subsequent pattern at the end of a current pattern. The LED flashes with every beat regardless depending on tempo.



References

StarUML, *Working with Statechart Diagram*, http://docs.staruml.io/en/latest/modeling-with-uml/working-with-statechart-diagram.html#uml-state, accessed 09/03/18

Stewart, Rebecca (2018), *State Machines* Lecture notes

Allain, Alex, *Bitwise Operators in C and C++: A Turorial,* https://www.cprogramming.com/tutorial/bitwise_operators.html, accessed 09/03/18