Project Report – Drum Machine

Name	Md Golam Tanvir Zim
Student No	180715305
Course	Real Time DSP (ECS732P)

Project Description

The objective of the project was to design a create a digital audio system which plays some fixed drum loops sequentially. The drums sounds were played in given patterns from preloaded buffers. The patterns and direction of playing was determined by the orientation of the Beaglebone Black Board with which an accelerometer was connected. The tempo of the drum sounds was controlled by a potentiometer and there was a Play/Stop button to start and stop playing the drum sounds.

Design Process and Implementation

The drum machine was designed and implemented in a number of steps.

Step 01: Playing single drum using single read pointer

In step 01, the drum sounds were played individually using a single ReadPointer. In this step, there was one pointer `gReadPointer` which would keep track of the samples of a drum sound played in each frame. The `gReadPointer` was incremented from zero to the length of the drum sound played, and then reset to 0. The lengths of the drum sounds were stored in an array `gDrumSampleBufferLengths[index]`, whose each element would store the length of drum sounds in the following manner.

`gDrumSampleBufferLengths[index]`

Index	0	1	2	3	4	5	6	7
Value	Length of Drum 0	Length of Drum 1	Length of Drum 2	Length of Drum 3	Length of Drum 4	Length of Drum 5	Length of Drum 6	Length of Drum 7

Step 02: Playing multiple drums using multiple read pointers

In step 02, multiple drums were played at the same time. One read pointer is needed to keep track of one single drum sound, so in this project for playing 16 drum sounds, 16 read pointers are needed, which would keep track of the samples being played in each drum. These 16 read pointers are declared in an 1D array of 16 element in the following format.

`gReadPointers[index]`

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Pointer Position	Initialized to 0	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..

For proper implementation of multiple drums using multiple pointers, another array, `gDrumBufferForReadPointer[16]` was required which would keep track of which read pointer is being used to play which drum. If a read pointer was used to play a drum, the corresponding element of the `gDrumBufferForReadPointer[]` would store the drum number. If a certain read pointer was not in use, then the corresponding element of `gDrumBufferForReadPointer[]` would store -1.

For instance, we can imagine a scenario, three buttons were pressed one after another to play drum 2, drum5 and drum 7 were being played before none of them are finished, then the read pointer array and the read pointer tracker array could store values as follows (random values are used here):

`gReadPointers[index]`

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Pointer Position	431	301	109	0	0	0	0	0	0	0	0	0	0	0	0	0

`gDrumBufferForReadPointer[index]`

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Pointer Position	2	5	7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Whenever a new drum sound was required to play, by looking at the values in `gDrumBufferForReadPointer[]`, (whether its -1 or not) the drum sounds would be assigned to a particular pointer. After the sound play was finished, the pointer and the pointer tracker both were released, by setting the values to 0 and -1, respectively.

This start of playing drum sounds was controlled using push button at this stage. Whenever a Button was pressed, `startPlayingDrum(int drum_index)` function was called, this would check which pointer is free and if there is any, then allocate the pointer for it and the drum was played.

Step 03: Playing single drum in loop with varying interval and Led blink after each loop

In this step, a single drum was played in loop, i.e, in a repeated order. This time, a button press would start and stop playing. To do that, a metronome counter was used. The metronome counter would start from zero to a certain value and then it would reset itself to zero. The counter value was calculated using the following equation.

$$\text{Metronome Counter Value in Sample} = \frac{\text{Metronome Counter in Milliseconds} * 44100;}{1000}$$

When a metronome counter reaches the highest value, then it would start playing a drum sound using the function `startNextEvent()`. At this stage, this function would call the `startPlayingDrum()` function with a random drum number. Later this function was changed.

The metronome counter, after reaching its counter value, would check if a button was pressed while playing by checking the variable `gIsPlaying` (Play = 1, Stop =0). If yes, then it would no more call startNextEvent(), unless button was pressed again as Start. In this way, the Stop button would not cut off the drum, rather the drum sound was played full and then stopped. Thus Play/Stop button was implemented.

The Led was blinked with each metronome beat. The Led ON time was set to 80% of the metronome counter value, so that LED blinking remains visual.

In the last stage of this step, a potentiometer was used to vary the metronome interval. A potentiometer was connected to the analog pin of Beaglebone Black. The metronome counter interval in milliseconds were varied according to the voltage of potentiometer. The analog reading value of Beaglebone is [0~1] which was converted to [50~1000], i.e. 50ms to 1000ms.

Step 04: Playing a single pattern of drums in loop

In this step, a single drum pattern was played. The patterns were stored in a array `gPatterns[][]` in the following manner:

Drum Index / Pattern Index	Event	Event	Event	Event	Event	Event	
0	drumX	drumX	NoDrum	NoDrum	drumX	drumX	…
1	NoDrum	drumX	drumX	NoDrum	drumX	NoDrum	…
2	drumX	drumX	NoDrum	NoDrum	NoDrum	drumX	…
3	…	…	…	…	…	…	…
4	…	…	…	…	…	…	…
5	…	…	…	…	…	…	…

This is an illustration of how the patterns are stored in the two dimensional array. The first element of the array contains the patten number. In this project we used 5 patterns (0~4). And the second elements of the array contain the drum number contained by the events in successive order.

The core points in this step are three.

- Initially, the current pattern to be played is defined by the variable gCurrentPattern (which we defined in this stage, later on defined by the orientation of the board using accelerometer.)
- The gCurrentIndexInPattern holds the value of the event in which a drum to be played (if exist) or not (if not exist).
- The eventContainsDrum() function tells if a given drum exists in a given event.

When each even is triggered, a for loop was used to check each drum in `startNextEvent()` function, whether they exist in that pattern. If it returns 1 then that drum exist in that event of the pattern being played.

On the above illustration,
drumX indicates a drum number, one of the drums (0~7) exist.
NoDrum indicates, no drum exist in that event, so nothing to play.

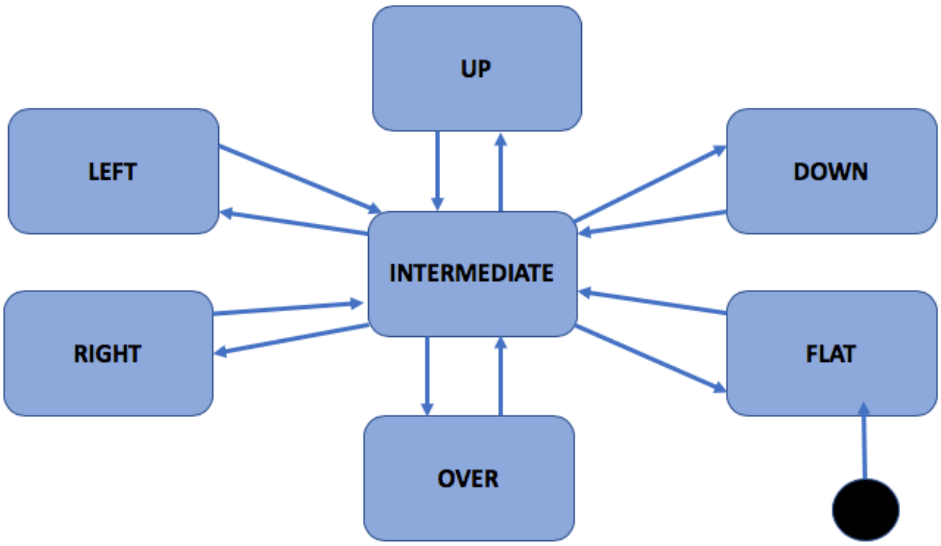
Step 05: Playing multiple pattern of drums defined by the orientation of the board in loop

In this step, the pattern in which the board to be played was defined by the orientation of the board. The orientation of the board was determined using an acceleroemeter.

Accelerometer gives three analog values on three different axis, representing how much acceleration is in effect in each axis. The analog value it produces in its three pins (x, y, z) are within the range (0~3.3V), which is translated into beaglaboard board as (0~1).

(0V~3.3V) range represents the range of acceleration (-1.5g ~ +1.5g) on each axis.

A state machine was defined to determine on which orientation the board is. They are FLAT, OVER, RIGHT, LEFT, UP, DOWN and INTERMEDIATE. The following diagram depicts the state machine.



For first six orientation, except the OVER state, the current pattern `gCurrentPattern` was be changed. For FLAT, RIGHT, LEFT, UP and DOWN states pattern 0, 1, 2, 3 and 4 were used, respectively.

Based on the readings taken from accelerometer, the seven states are detected as follows:

Range of Values	Orientation State
0.52<z<0.7	FLAT
0.15<y<0.25	OVER
0.55<y<0.7	RIGHT
0.20<y<0.3	LEFT
0.55<y<0.7	UP
0.20<y<0.3	DOWN
Else	INTERMEDIATE

The values were set in a way so that transition takes place between the first six states through INTERMEDIATE state. There are two key things

- Whenever the system enters into the INTERMEDIATE state, it resets the gCurrentIndexInPattern to 0, because a new pattern to be played.

- Whenever the system enters from INTERMEDIATE to any of the five states, it changes the global variable `gCurrentPattern` (0~4) according to the orientation.

Step 06: Playing backwards

When the board is made upside down, the state machine enters into OVER state. When it enters into that state, the currentPattern is not changed, however, the index of that pattern was set to 0. To play backwards,

- A variable was set to decide whether the drum should be played forward or backward (Forward, if `gPlayBackwards` = 0; backward if 1).
- If its in backward mode, the `gReadPointers[x]` which will play the drum has started played the drum from the end of the buffer back to zero, instead of zero to end of the buffer in forward mode.
- If the state machine enters exits the OVER state, the Index is set to zero and the mode is set as forward mode.

Limitations:

1) There threshold value for which the orientation change took place could me more accurately defined. The state machine was designed in a way so that the transition from one state to another would occur through intermediate state. But, the board position might be changed very quickly so that it doesn't pass through intermediate state, in that case, the pattern will not be properly. To get rid of that or any wobble effect, hysteresis loop can be used to make smooth transition through strictly defined path.
2) The Led should turn on those beats when there is a drum available. But in this project, led was turning on with every event being triggered. This could be solved by using a variable which would be set to 1 if a drum exist in the given event. If no drum exist, then the Led will not turn on.