15-323/623 Spring 2019

Project 1: MIDI Drum Machine

Due Jan 29

1. Overview

In this project, you will build a MIDI drum machine using *Serpent64 or wxSerpent64* to gain some basic knowledge about real-time music systems and (relatively) accurate timing. As with a real drum machine, your implementation should provide a flexible system with adjustable parameters. This project will act as the basis for the future projects.

2. Specifications and Implementation Guide

Specifications:

Given a pre-defined *tempo* (beats per minute) and a repeat count *N*, either from console[1] or through a (simple) graphical interface, the program will play *N* repetitions of a 4-beat drum pattern. The pattern should be table-driven and consist of at least three separate drums, e.g. high-hat, bass drum, and snare drum. Use General MIDI to determine the MIDI channel(s) and note numbers. Even though drum sounds typically decay quickly, send an explicit note-off (key-up) message for each note-on (key-down) message.

You are encouraged to make your own drum pattern, but you may use this one:

Beats(s)	1		2		3		4		1		2		…
High-hat	X	X	X	X	X	X	X	X	X	X	X	X	…
Snare Drum			X				X				X		…
Bass Drum	X				X				X				…

Implementation guide:

1. Correctly configure MIDI before you start: Don't expect sound from your program unless MIDI is configured. To test you MIDI configuration, you can use the function *midi_example(),* which is defined in `http://www.cs.cmu.edu/~music/serpent/doc/serpent-by-example.htm`.

2. Do not use `sched.srp`, `midi-io.srp`, threads, or other libraries: We'll use higher-level interfaces later, but this project is intended to make you think about the challenges of concurrency and timing. While you can pass timestamps to PortMidi, *do not use PortMidi timestamps to schedule or sequence or time your output.* Wait until it is time to send a MIDI message and then send it with timestamp zero (0), which means "immediately."

3. Learn about MIDI in Serpent:

Please see `http://www.cs.cmu.edu/~music/serpent/doc/serpent-midi.htm`

4. Never use busy-wait for timing: One possible implementation is to have a busy-wait loop, repeatedly testing if it is time to send the next message. However, this is not acceptable. The CPU load should be low when running your program. Hint: the function *time_sleep(sec)* is your good friend for this. Also, *time_get()* will tell you the current time, which is useful.

[1] In serpent, you can use `print "prompt:",` to prompt and `stdin.readvalue()` to read values.

5. Real/virtual time conversion: Real time (measured in minutes/seconds) and virtual time (measured in measures/beats) are somewhat confusing in time computation; please be careful. Remember they are connected via tempo (beats per minute). Feel free to ask the course staff if you are unclear about the definition of music terms.

6. PortMidi Device Number: Ideally, PortMidi's *midi_out_default()* call will give you a good device number for output. In fact, if it doesn't, you can use the `PmDefaults` application (part of PortMidi) to choose the default device. If you have problems with that, you can define a global that can be easily found and changed, e.g. `MIDI_OUT_DEVICE = 3`, or you can *prompt* for an output device number after printing device options (as in *midi_example()*). To read a number, you can use `stdin.readvalue()`.

3. Grading Criteria:

1) Correctness: Does the program compile, run, and produce the desired output? Are the number of measures and tempo configurable?

3) Accurate timing: Do the drums sound steady and musical? If you play 1001 beats at 120 bpm, is the last beat exactly 500 s after the first beat (good), or do small timing errors accumulate (not so good)?

2) Modularity: Your code should be clearly organized into functions for initialization, execution, and finalization, and you should provide clean interfaces to access data.

3) Programming style: Code should be clearly written and commented to optimize readability (include high-level design and specifications at the top of the file or in a separate document, give concise comments within the code, avoid verbose or redundant inline comments.) A 2-page instruction manual in Latex is overkill, but a program with 10 lines of comments needs work. Note that indentation in Serpent is significant (like Python), tab characters are *not* acceptable, and 4-space indentation is required for this class.

4) Drum pattern representation: Your program should make it easy to change the drum pattern. The drum pattern should be represented as data rather than hard-coding a sequence of MIDI send commands and delays.

4. Hand-in Instructions:

Hand in your project to Autolab (`autolab.andrew.cmu.edu`) system – note the URL has no "cs" in it – as a .zip file containing the following things:

1. source code
2. additional documentation files, if any

To grade your project, we will:

1. unzip your directory
2. cd to your directory
3. run serpent64 or wxserpent64 with no arguments

5. Additional Requirements for 15-623 Students (also extra credit for 15-323 students)

1) 15-623 students should use wxserpent64 to implement a graphical interface.

2) There should be some kind of default pattern or a way to select from a set of built-in patterns.

3) There should be a way to enter new patterns. At minimum, this can be some kind of ASCII-based pattern using a *Textctrl* objects or a *Multitext* object, but it should not be too hard to create an array of *Checkbox* objects or even draw your own grid using a *Canvas*.

4) There should be a *Play* button and a *Stop* button.