*TeensyAudio Wavetable Synthesis*

Software Verification and Validation Plan

|  |  |
| --- | --- |
| Authors: Ryan Mellmer, Connor Delaplane | Date: 2/9/2017 |
|  | |

# Introduction

## Purpose

The purpose of this document is to outline how the TeensyAudio Wavetable class and accompanying SoundFont decoder utility will be verified as “correct” and fulfilling the software’s functional and technical needs.

# Testing Approach

The AudioSynthWavetable C++ library will be validated using two “Test Sketches”, containing code to thoroughly test the following functionality on the Teensy for correctness in pitch-interpolation and enveloping as well as benchmarking tests for performance. These sketches will be uploaded to the Teensy, and test results will be printed to the Arduino Serial Monitor.

The first sketch is called Unit\_Interp and is responsible for testing correctness of the interpolation algorithm. The second, Unit\_Main, performs all other tests. These tests can easily be performed selectively by simply commenting out #define [test] statements corresponding to tests that a user does not wish to perform. The interpolation test is kept in a separate sketch because it is memory- and processor-intensive.

## Component Testing

* + 1. **Interpolation**

In order to test our interpolation functionality, we will develop a test sketch called Unit\_Interp which will instantiate an instance of the AudioSynthWavetable class, and iteratively play a range of different notes using a simple sine wave as the input to the wavetable object, and then check that the output frequency is the expected frequency for whatever note we triggered, within some allowed margin of error. Using the YIN fundamental frequency estimator algorithm as implemented by Colin Duffy within the AudioAnalyzeNoteFrequency class, we are able to analyze the raw audio output frequency with high accuracy, provided the input waveform is simple. Therefore, we will be using a simple sine wave as the input “instrument” sample to the AudioSynthWavetable object because this is the simplest waveform to approximate output frequency for, and will produce the most accurate result. Results of each note test will be printed to the serial monitor. Final test results will be displayed after running all individual note tests.

The frequency analyzer object only works reliably (given an appropriately simple input waveform) for frequencies between about 0.029 and 4.0 kHz, corresponding to MIDI notes 22 and 107, respectively. As such, we will only test notes in this range. Furthermore, for ease of automated testing, the algorithm only tests notes in the western scale (12-note octaves). This corresponds exactly to the standard MIDI range, so this suffices for our purposes.

**Test Pseudocode**

The following is high-level pseudocode for the basic operation of Unit\_Interp.

for (each note in [22 - 107]) {

input\_frequency := frequencyOf(note)

playNote(note)

while (note has been playing for less than 500 milliseconds)

output\_frequency := getOutputFreq()

if (|input\_frequency - output\_frequency| <= TOLERANCE)

tests\_passed += 1

}

print (tests\_passed / total\_tests)

* + 1. **Enveloping**

To test the enveloping functionality, we will include an envelope subtest within the Unit\_Main test sketch (defined as TEST\_ENV). This test will play a sample using the wavetable object, and check the amount of time (in ms) spent in each envelope state against the expected values read from the decoded sample data. Section lengths will be tested for several different MIDI notes between 0 and 127, with the goal of testing multiple samples for each instrument, as section lengths can vary between different decoded samples of a given instrument (with different note ranges). Results for each test will be displayed via the serial monitor.

Any sample instrument can be used for this test. However, tests run on samples with at least one “long” section, and/or instruments with multiple samples that have differing section lengths, are most informative. For this test, we only care about how much time the code spends in each state of the envelope, as this is the largest bug/issue we have come across and had to fix within the enveloping code.

**Test Psuedocode**

The following is high-level pseudocode for the basic operation of TEST\_ENV.

for (each decoded sample of an instrument) {

play a note within the range of that sample

reset timer

current\_state := wavetable.getEnvState()

while (wavetable.getEnvState() != STATE\_IDLE) {

if (current\_state != wavetable.getEnvState()) {

compare timer value with the expected section length

if (|timer - expected\_length| < TOLERANCE)

tests\_passed += 1

reset timer

current\_state := wavetable.getEnvState()

if (current\_state == STATE\_SUSTAIN) {

wavetable.stop()

current\_state := wavetable.getEnvState()

}

}

}

}

print (tests\_passed / total\_tests)

## Performance Testing

* + 1. **CPU Usage**

To test CPU performance of the wavetable class, we will include a CPU usage subtest within Unit\_Main (defined as TEST\_PROC). This test will incrementally trigger more and more AudioSynthWavetable objects to play on different notes. The result will be that many different objects will eventually be playing concurrently. As each voice is added, processor usage is computed and printed to the serial monitor. Testing will stop when either the 64th voice is added or processor usage surpasses 95%. Test results will be used to determine our final benchmark numbers for the AudioSynthWavetable class.

**Test Psuedocode**

The following is high-level pseudocode for the basic operation of TEST\_PROC.

count := 0

timer := 0 // Increments by 1 every millisecond

note := start\_note

while (count < 64) {

wavetable[count].play(note);

while (timer < timer\_period) {

}

proc\_usage = getProcUsage()

print (proc\_usage)

if (proc\_usage > 95%)

break loop

else

tests\_passed += 1

count += 1

note += 1

timer := 0

}

**Latency**

* + 1. Based on our research regarding perceptible audio latencies, our goal is to keep latency (time from note trigger to audio output) under 10 ms. In order to validate this, we will add a latency subtest to Unit\_Main (defined as TEST\_LATENCY). This test will periodically trigger playNote calls for different notes on an AudioSynthWavetable object and measure the elapsed time in ms between function call and when audio output data is detected by an AudioAnalyzePeak object. This object is only used for detecting audio output; it is not used for actual peak analysis in our context. This suffices for our purposes.

**Test Psuedocode**

The following is high-level pseudocode for the basic operation of TEST\_LATENCY.

timer := 0 // Increments by 1 every millisecond

count := UPPER\_BOUND

while (count != LOWER\_BOUND) {

timer\_latency := 0 // Increments by 1 every millisecond

wavetable.playNote(count)

when (peak\_detector detects audio data) {

latency := timer\_latency

print(latency)

if (latency ≤ TOLERANCE)

tests\_passed += 1

count -= 1

}

while (timer < timer\_period) {

}

timer := 0

}

## Regression Testing

To perform regression testing, we will run the above mentioned test sketches before each major release to ensure that all of the general requirements for the system hold, including interpolation correctness, enveloping correctness, and ensuring that performance is equivalent to or better than the previous iteration. We will also add any additional “checks” to the above testing sketches upon finding specific bugs/issues that can be identified automatically.