I'm singing this note 'cause it fits in well
With the chords I'm playing
I can't pretend there's any meaning here
Or in the things I'm saying

But I'm in tune
Right in tune

-- The Who, "Getting In Tune"

Introduction

For our seventh and final assignment, we will move into the realm of analog signals. We will explore digital-to-analog conversion (DAC) and analog-to-digital conversion (ADC) using the KL25Z's direct memory access (DMA) subsystem. You will find that, when dealing with analog signals, the finicky details of math really matter: We are aiming for a respectable level of accuracy.

As before, you will develop a proper GitHub project for this assignment, complete with code, documentation, and README. Details are given below.

Assignment Background

You just got a text message from your friend Pete Townshend, the legendary guitarist and songwriter. Pete would like you build him a next-generation guitar tuner based on the KL25Z. Since you're an otherwise down-on-your-luck masters student, you get right to work on a prototype. (Pete is the guy who wrote the lyric "Yeah, I hope I die before I get old"—and he's now 76. Really, there's no time to waste!)

We are going to use the KL25Z to mathematically generate four pure tones in the form of analog signals. We will play the tones out using the DAC, and then sample the resulting analog output using the ADC. Fortunately, it is possible to configure the KL25Z so that the DAC output becomes an ADC input.

We will use the following parameters:

	Output (DAC)	Input (ADC)
Sampling rate	48 kHz	96 kHz
Resolution	12 bits per sample	16 bits per sample

Your code should cycle through the following musical notes: 440 Hz (an A4); 587 Hz (a D5); 659 Hz (an E5); and 880 Hz (an A5). Each note should be played for one second. When playback reaches the end of this list, it should restart at the beginning.

Once a note is playing, you should use the ADC to read 1024 samples, and then you should perform some relatively simple analysis, outlined below.

You will need to develop the following:

1) **A function to accurately compute sin(x) using only integer math**. You may do this using either a lookup table or a hybrid Taylor series approach, as long as you meet the requirements below.

2) **A tone_to_samples() function**, which computes the samples representing a pure tone of a specified frequency, based on your sin(x) function. Use this function to fill a buffer of output samples.

3) **An audio output module**. This module will contain the necessary configuration and runtime code that allows you to pass your computed buffer of audio samples into the module. The module will use the KL25Z's DAC, TPM0, and DMA hardware to repeatedly play out the buffer without further intervention from your main loop.

4) **An audio input module**. This module will contain the necessary configuration and runtime code to capture a number of input samples at the capture sampling frequency specified above.

5) **An audio analysis module**. After capture, your code should analyze the captured buffer, reporting the minimum, maximum, and mean values. You should also integrate my autocorrelation function, provided on Canvas, to recover the fundamental period represented in the sampled audio.

6) **An oscilloscope.** After you have all the code working, you should connect an oscilloscope to the DAC's output on pin PTE30, available on the Freedom header at J10 11. Use the oscilloscope to verify that there is no tearing at the transition point in your playout buffer.

Implementation Details

As in our previous assignments, this project is intended to be a bare metal implementation, so you may not use advanced SDK routines. You may use "MKL25Z4.h". You may not use the "fsl_" include files that provide higher-level SDK-based functions, with the exception that you may use "fsl_debug_console.h". You may use printf() and similar C library functions.

Your integer sin() function should be able to take any arbitrarily scaled input and produce the corresponding (scaled) sin(x)[1]. You should test your sine function using the following code on your PC:

```
void test_sin()
{
  double act_sin;
  double exp_sin;
```

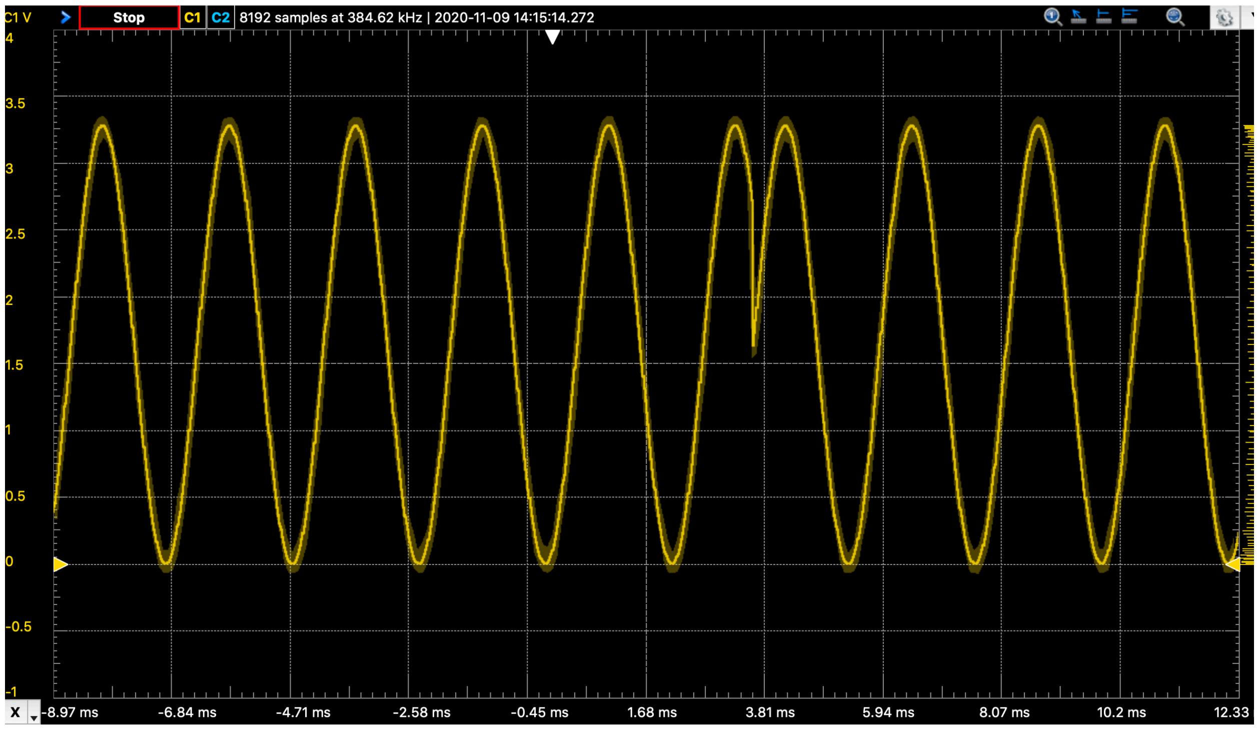
[1] In other words, your function should accept input in the range [INT_MIN, INT_MAX] and should produce output in the range [-SCALE_FACTOR, SCALE_FACTOR] for whatever scale factor that you select.

```
    double err;
    double sum_sq = 0;
    double max_err = 0;

    for (int i=-TWO_PI; i <= TWO_PI; i++) {
      exp_sin = sin( (double)i / TRIG_SCALE_FACTOR) * TRIG_SCALE_FACTOR;
      act_sin = fp_sin(i);

      err = act_sin - exp_sin;
      if (err < 0)
        err = -err;

      if (err > max_err)
        max_err = err;
      sum_sq += err*err;
    }

    printf("max_err=%f  sum_sq=%f\n", max_err, sum_sq);
  }
```

Your code will need to provide the #defines for `TWO_PI` and `TRIG_SCALE_FACTOR` and the function `fp_sin()`. Before proceeding, ensure that your maximum error is under 2.00, and your sum-of-squares error is under 12000. Except for the tests in this function, we only use integer math for this assignment.

I want to emphasize the importance of implementing the sine function using only *integer math.* If we were going to use floating-point operations, there is a perfectly good sin() function available in the standard math library!

Remember that the buffer you compute with the output waveforms (that is, output samples) will be played repeatedly by your audio output module. Therefore, when computing this buffer, take care to ensure there is no tearing at the wrap point. To illustrate, the behavior shown in this oscilloscope trace is a bug:



Your analog output module should use DAC0, DMA0, and TPM0. You will need an interrupt handler to handle the wrapping around the buffer. See chapters 6 and 9 in Dean for guidance. Note that unlike Dean's code, you will need to copy aside the samples handed in from higher-level code before beginning playback from your private buffer. (You do NOT need to worry about tearing at the boundary between two different notes.)

Your analog input module should use ADC0 to read samples in a polling loop. Because software methods to trigger the reading of each sample are highly susceptible to jitter, you should use TPM1 to trigger your reads. (Remember, *Pete Townshend* is going to be using this code!)

Each time through the main loop, you should print a message similar to the following via the UART:

```
Generated 981 samples at 440 Hz; computed period=109 samples
min=0 max=65535 avg=34079 period=216 samples frequency=444 Hz
```

The first line of text is written after the output buffer is computed.

The second line of text is written after getting the output playing, then reading in the samples using the ADC, and then analyzing them. The period (in samples) is different because the sampling rate is different, but the observed frequency of 444 Hz matches the desired frequency relatively well.

All periods are reported here in units of samples/cycle, and not the more common time/cycle that you are likely familiar with.

Submission Details

Your submission is due on Tuesday, November 30, at 10:30 am. At that time, you should submit the following to Canvas:

- A private GitHub repo URL which will be accessed by SAs to review your code and documentation. For this assignment, this repo should have an MCUXpresso project containing multiple .c and .h files. **Please check the MCUXpresso tree directly into GitHub**—do not zip it or extract individual files from it.
- A PDF containing all C code. As before, the PDF is used specifically for plagiarism checks.

Grading

Points will be awarded as follows:

Points	Item
1	Did you submit your GitHub URL in your Canvas submission?
24	Overall elegance: Is your code professional, well documented, easy to follow, efficient, and elegant? Is each module (your trigonometry function, analog_out, analog_in) located in its own file, with an appropriate .h file to expose the interface?
25	Is your sin(x) function implemented correctly, **using only integer operations**[2]? Does it meet the requirements stated above (max error and sum-of-squares error)?
25	Does your analog output module work as specified, with the appropriate sampling rate and other parameters? Are waveforms computed and generated properly? Is there tearing at the buffer wrap point?
20	Does your analog input module work as specified, with the appropriate sampling rate? Do you perform the analysis on the input as outlined above? Does the observed input frequency roughly match the computed output frequency?
5	Connect your DAC output to an oscilloscope. Validate your output waveforms, and in particular ensure there is no tearing at the transitions. Submit a screenshot showing each of your four waveforms.
5	Extra credit: Implement the ADC calibration routine specified in the KL25Z Reference Manual, §28.4.6.

Good luck and happy coding!

[2] Use of floating-point operations will result in a zero for this section.