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 STM32'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

We are going to use the STM32 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 STM32 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	12 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 **two seconds**. 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**. This function is already provided to you and the example use is given in test_sine function.

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 STM32 DAC, TIMER6, 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 in the github, 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 PA4, available on the CN8 header pin 3 as well as CN7. Use the oscilloscope to verify that there is no tearing at the transition point in your playout buffer.

Implementation Details

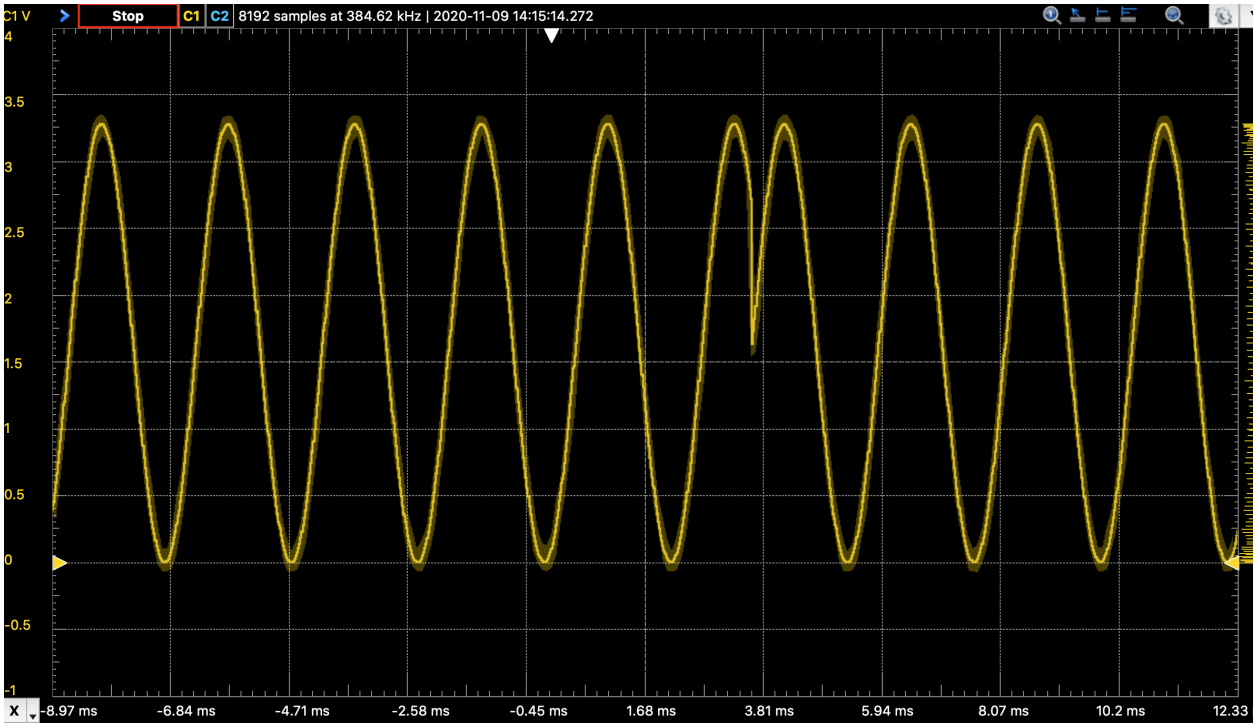
As in our previous assignments, this project is intended to be a bare metal implementation. You have been provided a base project in github as a starter and the clock rates are given in the main file. You may use printf() and similar C library functions including any multiplication and division operator.

Following test_sin function illustrates how to call the integer sin function from the library:

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

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 DAC, DMA, and TIMER6. 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 starting the playback from your private buffer. You should generate enough waveform samples to fill as much of 1024 output buffer as you can as discussed in the lectures. (You do NOT need to worry about tearing at the boundary between two different notes.)

Your analog input module should use ADC 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 TIMER1 to trigger your reads.

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 when you start playing the output, reading the samples using the ADC, and then analyzing them. As you can see, 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 27, at 9:00 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 STM32CubeIDE project containing multiple .c and .h files.

Grading

Points will be awarded as follows:

Points	Item
30	Overall elegance: Is your code professional, well documented, easy to follow, efficient, and elegant?(15) Is each module (analog_out, analog_in and others) located in its own file, with an appropriate .h file to expose the interface?(15)
30	Does your analog output module work as specified, with the appropriate sampling rate and other parameters?(7.5) Are waveforms computed and generated properly?(7.5) Is there tearing at the buffer wrap point?(7.5) Is the peak to peak voltage at least 3.1V?(8). Basically 2.5 pts for each waveform.
30	Does your analog input module work as specified, with the appropriate sampling rate?(10) Do you perform the analysis on the input as outlined above?(10) Does the observed input frequency roughly match the computed output frequency within 5 Hz?(10 pt if within 5 Hz, 5 pt if within 10 Hz, 0 otherwise)
10	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 and the frequency, peak to peak V.(2.5 pts for each waveform)

Good luck and happy coding!