Lab 3: guitar tuner

In this lab exercise you will practice with ADC. To be able to go through this lab you will need to have knowledge related to:

- General C programming.
- The ESP32 in general and its ESP-IDF software development kit.
- Some basic digital input/output using GPIOs.
- Analogue to digital conversion.

In terms of materials you will need:

- A NodeMCU 32S board or equivalent.
- USB cable.
- A breadboard.
- One pushbutton.
- A bunch of wires.
- One amplified microphone.
- 2 LEDs

The device works like this: starts from one note, corresponding to the thinnest of the strings, one plays a note on the guitar (or any instrument! It can be your voice too) and the device indicates if the played note is higher (one LED flashes), lower (the other LED flashes) or OK (both LEDs flash) compared to the note it is supposed to play. When the user presses the button, the tuner sets the goal note to the one corresponding to the next string.

The standard notes for each string are:

String	Note	Frequency (Hz)
1 (the thinnest)	E4	329.63
2	B3	246.94
3	G3	196.00
4	D3	146.83
5	A2	110.00
6	E2	82.41

See here and here for more details.

As with lab 1, we will go through this step-by-step.

Circuit

Microphone board:

- AR: floating
- OUT: connect it to pin 32 or 34
- GAIN: floating
- VDD: connect it to 3.3V (there's a pin for that on the NodeMCU board)
- GND: connect it to GND on the NodeMCU board

LEDs:

- Connect one between GND and pin 12
- Connect the other between GND and pin 13

Button:

- Between GND and 18 (or any other input pin)

Sampling a signal

Here we want to gather a sound and understand what the main frequency of this sound is. First things first, you will need to **sample** your incoming audio signal. Think about it: at what frequency do you need to sample it in order to avoid aliasing? Please consider that your microphone is a cheap one and it won't pick anything beyond 10kHz.

Once you have decided at what frequency you sample the signal, you need to setup a strategy for acquiring and analysing the signal. You probably want to take a sample using the ADC every given number of milliseconds, so you'd probably want to use **timers**.

Remember that there are two types of timers in the ESP-IDF, the hardware timers (more powerful, but more cumbersome to setup) and the software timers (less accurate, but simpler to program). Now the choice depends on the sampling frequency you want to achieve. According to the documentation, software timers are less precise and have a minimum period of 50 us (or maximum frequency of 20000 Hz). If you are happy with that, then you can use them.

If you want to use hardware timers, remember that you can choose to the ISR loaded into RAM or ROM. If you choose RAM; then you should not use functions from the ESP-IDF, including the ones accessing the ADC converter! You would need to access the registers directly. So my suggestion is to load the ISR into ROM instead, also because you don't need too much speed.

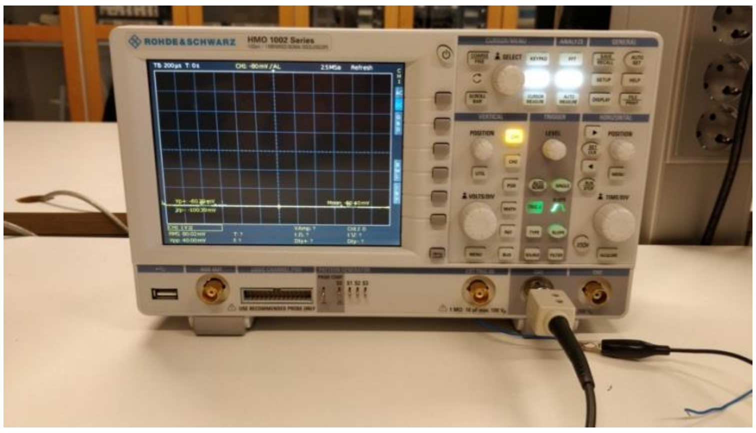
In summary the idea is that, using a timer every X microsecond (depending on your chosen sampling frequency), the timer callback or ISR should read a value from the ADC.

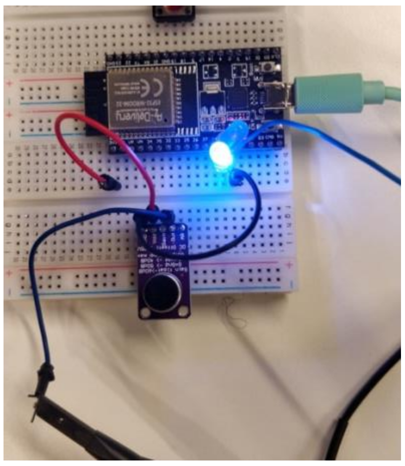
Let's start structuring the code, create a little module like this one (sampler.h):

```
#ifndef SAMPLER_H_
#define SAMPLER_H_

/**
 * Starts sampling on a given pin with a given fequency.
 * Parameter freq: the sampling frequency
 */
void startSampling(int freq);

/**
 * Stops the sampling process.
 */
void stopSampling();
#endif
```

And place the timer with its ADC reading in its corresponding C file. Refer to the lectures about timers and ADC as a reference.

Once you have your timer reading the ADC at your chosen frequency, it's time to test it. To understand how the signal looks like, let's use the oscilloscope.

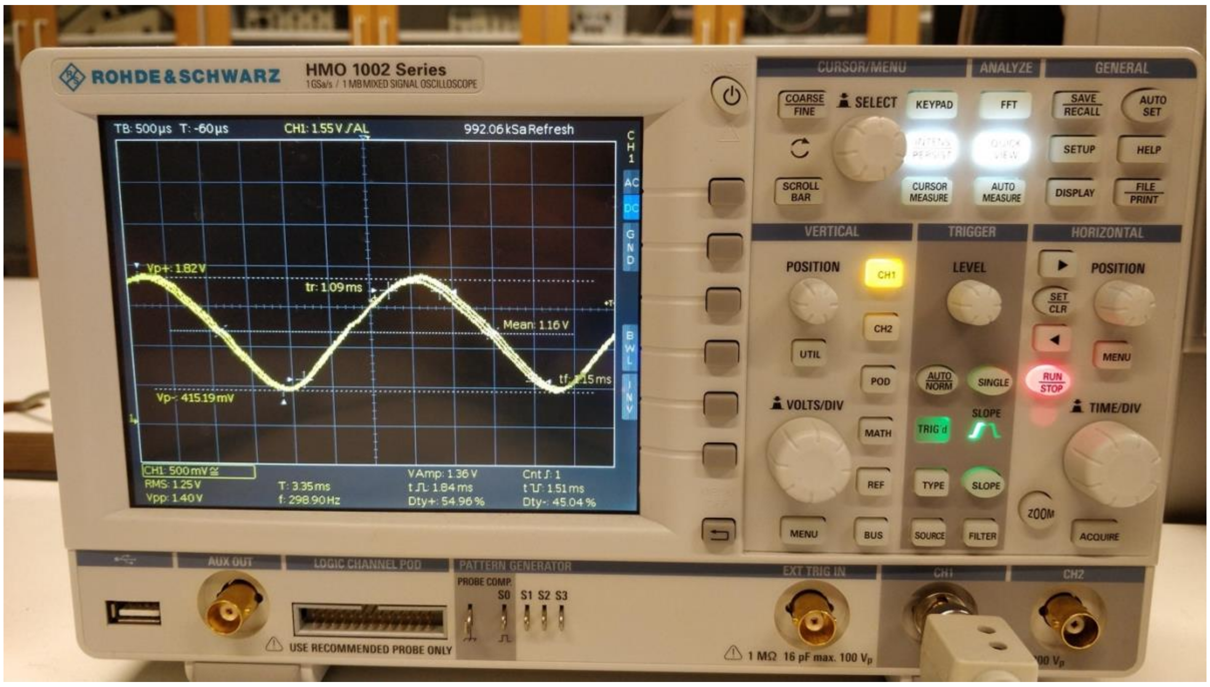




First off, connect the probe to one of the 2 available channels on the oscilloscope. Then connect the GND probe to GND on the board and the other probe to the output pin of the microphone. Say something, or make some noise, can you see the waveform?

Now let’s test a specific note. Using your phone, or the speakers of your laptop, play a tone at a specific frequency. You can use this a tone generator like [this one](). Can you measure the frequency (or the period) on the oscilloscope?

Use the vertical and horizontal knobs in order to adjust the X and Y axes. You want to be able to see the full period of the waveform. Once you have a nice complete period represented, take a snapshot by pressing “STOP”:



Under the waveform, you should be able to see the period (T) and the frequency (f) of the signal. This is automatically computed by the oscilloscope. Does it match what you are playing?

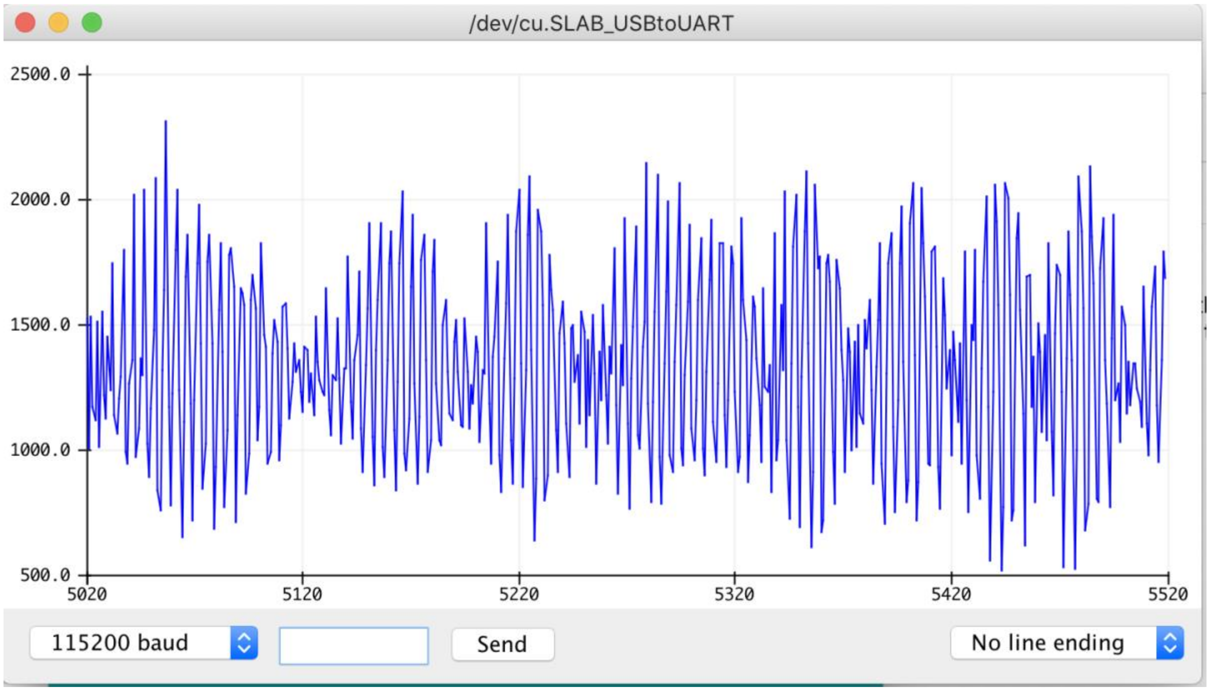
Play around with some examples, use your voice, sing or whistle, try with some instrument (or a [YouTube video]()?). It is important to get an understanding of how the oscilloscope works. It has many advanced features, but here we can only explore the basic ones.

If you want to know more, there are plenty of tutorials and videos online. [Here]() is a good one.

Once you have an idea of the analogue signal, let’s see what is actually read by the ADC. In the ISR (or software timer callback), print the value that is read by the ADC on the serial line, plus a new line. You can then use the Arduino IDE to plot the waveform, see a tutorial [here](), or you can try with [this VSCode extension]().

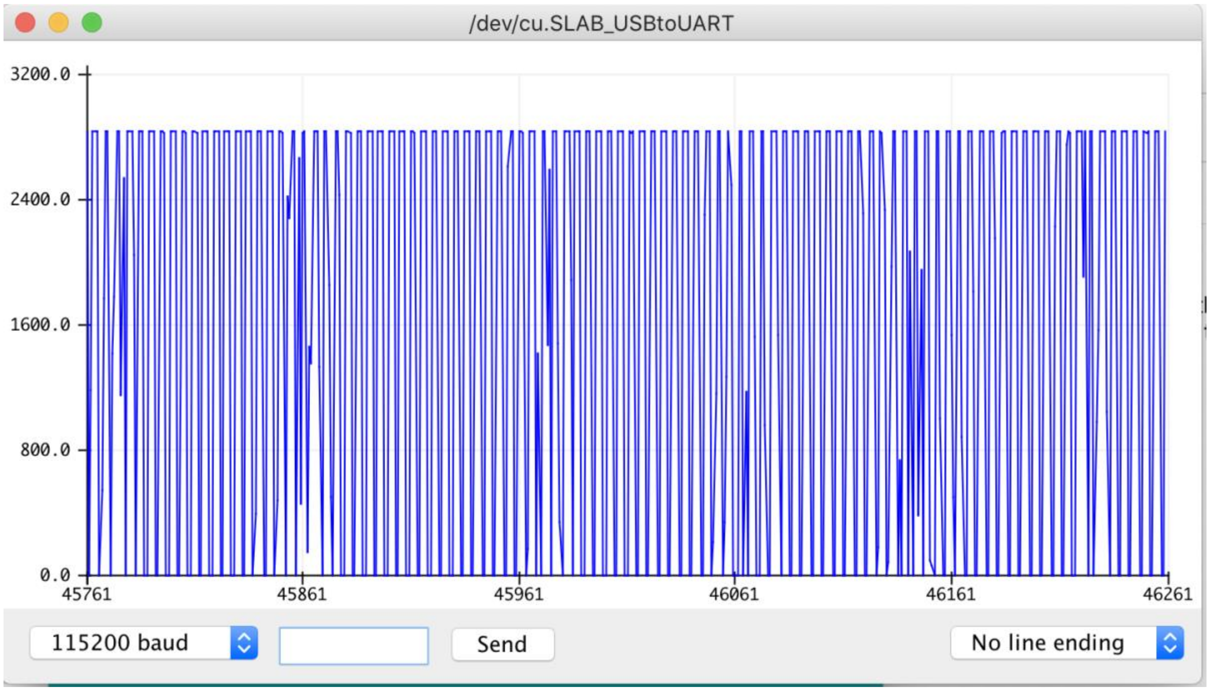
Note: sending data over the serial line at 115200 bits/s allows you to transmit numbers of 5 characters at a maximum frequency of about 2800 number/s. So, choose a sampling frequency of, say, no more than 2000Hz for this test.

I have run the test myself at 2000Hz and this is what I can see on the serial plotter of Arduino when I "sing" a low frequency tone into the microphone:



Which looks like a sound wave! However, it may be a bit difficult to analyse because of its variation (you'll see later why).

If put my mouth very close to the microphone I get:



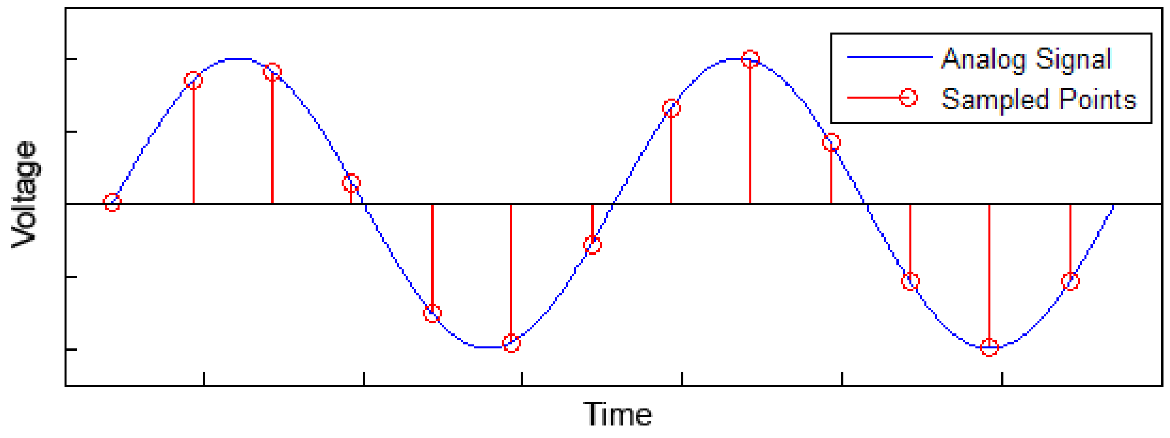
which is a "saturated" signal and will be easier to analyse later. Let's keep this in mind.

Detecting the frequency

We will make a brutal simplification here: we will assume that the sound we are recording has only one main frequency. This is OK as long as the signal we want to analyse is simple enough, which, as you have seen on the example before, can be obtained when the sound is loud and saturated. For example, when you put the source of the sound (your mouth, or your phone) close to the microphone.

OK, but how should you use those samples that you get in the ISR?

The idea is pretty simple: if you have a periodic signal, it will "cross" its average value two times during its period (or, equivalently, 2 x frequency times each second passed).



So, you could count the number of "crossings" and divide it by 2 x the duration of your signal to obtain the signal's frequency. A crossing is easily computed by detecting when the previous sample is below the average value and the next sample is above the average (or vice versa).

In terms of how you can detect those crossings over the whole signal, I'll give an idea: store the value of the last sample, and if the current sample is above the threshold and the previous is below the threshold (or vice versa). You can then divide the number of crossings by twice the total duration of the signal, which you need to keep track of, for example by storing the time when the first sample is retrieved.

Once you have thought about how to perform this "crossings" detection, it's time to program it. Let's add another function to our sampler.h module:

```
/**
 * Computes the average frequency of the signal that has been (or is being) sampled.
 */
float getFrequency();
```

You will need here to store some variables in this module (like the last sample that was taken): make sure they are only used within this module (can you remember how?) and that those that are used in ISRs are volatile.

An observation: you need to know what the average value of the waveform is in order to know when the signal crosses it. A sound wave has negative and positive values, with its average in 0, but the ADC can only measure positive voltages. To solve this, the microphone board adds a constant voltage to the signal, an offset. Look the microphone board closely: it is written “DC offset 1.25 V”. So the board actually moves what would be normally 0 mV to 1250 mV. The problem is that you need to know to what integer 1250 mV corresponds to, when read by the ADC. You could simply measure it experimentally, for example printing out or plotting the samples in complete silence, or just do the maths (it’s simple!).

To test if your code works, run a main like this:

```
#include <stdio.h>
#include <esp_task_wdt.h>
#include "soundgen.h"
#include "sampler.h"

void app_main()
{
   while (1)
   {
      printf("--- Start!\n");

      // sampling at 6kHz, max detectable freq = 3kHz
      startSampling(6000);

      // wait 1 second
      vTaskDelay(pdMS_TO_TICKS(1000));

      stopSampling();
      printf("--- Stop!\n");

      float fr = getFrequency();
      printf("Frequency is %.2f\n", fr);
   }
}
```

Then open a tone generator like this one and play it loudly with your mobile phone or the computer’s loudspeakers, while you observe the output over the serial console. In theory, the frequency you set on the sound generator should be the same that you read from your console (with a little error of course).

Given that starting and stopping the sampling takes a little time, we can maybe improve this by adding a function, let's call it:

```
/**
 * Resets frequency detection without stopping the sampling process.
 */
void resetSampling();
```

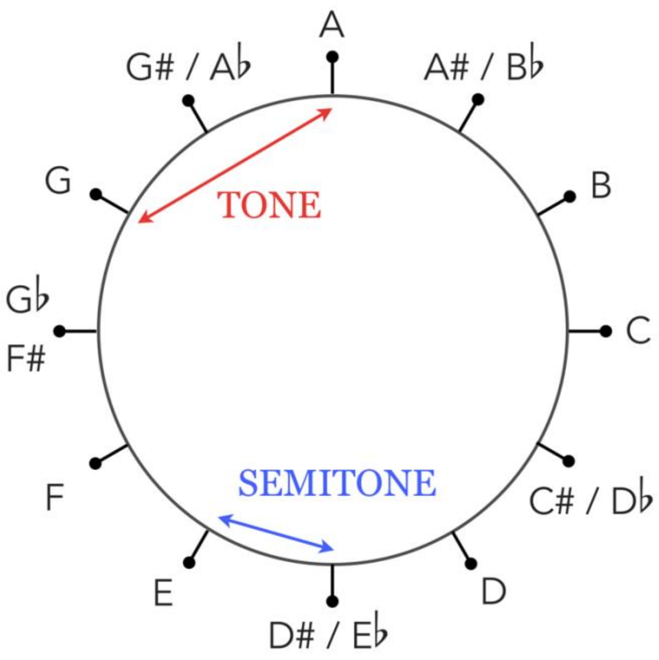
which basically resets the counter and any timestamp you are taking, without stopping the timer. So, in the main, instead of calling sartSampling() and stopSampling() in the infinite loop, you can simply call:

```
  startSampling(6000);

  while (1)
  {
    vTaskDelay(pdMS_TO_TICKS(1000));

    resetSampling();

    float fr = getFrequency();
    printf("Frequency is %.2f\n", fr);
  }
```

From frequencies to notes

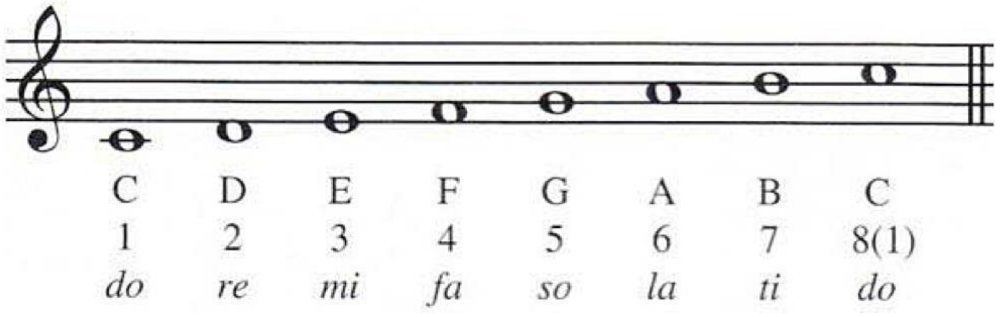
Instead of printing out the frequency you measure, it would be nice to print the note. The relationship between frequencies and notes is an interesting one, and you can read more about it here and here.

For our purposes, the number of semitones from **A4** is: n = 12*log2(f/440 Hz), where f is the frequency we are detecting. For those who need crash course on music, a semitone is the note very next to another one. So, say, from C to D# there are 3 semitones. See picture below:



A4 is just one of the several A (la) available, the one at 440Hz specifically. So if your frequency is, for example, 349.23, then the number of semitones from A is: -3.99, basically 4, which corresponds to F (check the wheel above).

Those, like me, used to syllabic musical notation (do, re, mi…), here is a useful translation:



We could then create a nice module for that (notes.h):

```
#ifndef NOTES_H
#define NOTES_H

/**
 * Converts a frequency to a note.
 * @param freq frequency, in Hz
 * @param stringbuff pointer to a string with space for 2 characters and null terminator
 */
void freq2note(float freq, char *stringbuff);

#endif
```

Be aware that log2 is available in the standard C library math.h. Pay attention to the fact that log2 gives you a float, and you want an integer. Should you just do a regular casting? Or would you rather use something like the round() function?

Also consider that n can be positive or negative.

Now try this out with something like:

```
void app_main()
{
   printf("--- Start!\n");
   startSampling(6000);

   while (1)
   {
       vTaskDelay(pdMS_TO_TICKS(1000));

       float fr = getFrequency();
       printf("Frequency is %.2f\n", fr);
       char note[] = "1234";
       freq2note(fr, note);
       printf("Note is %s\n", note);

       resetSampling();
   }
}
```

Play a tone with the tone generator (the interface allows you to select a note instead of a frequency) and check that it's working correctly.

Developing the user interface

Now that you know how to measure a frequency from the microphone, it's time to develop the primitive user interface. The output is just two LEDs, which you want to switch on and off, maybe flash. You know how to do this from lab 1.

Another important aspect here is that the device may be busy sampling the sound while the user presses the button. How do allow the button to stop the current execution? You need to use an interrupt of course!

Connect a button between GND and an input pin (I use 18). Then configure both the LED pins and the button pin. You can create a few functions for this, something like:

```
#ifndef PINS_H_
#define PINS_H_

#include <stdint.h>

/* initialises the LED pin */
void initLEDPin(uint8_t pinN);

/* initialises the button pin */
void initButtonPin(uint8_t pinN);

/* switches LED on if level!=0 or off if level==0*/
void setLED(uint8_t pinN, uint8_t level);

#endif
```

The LED functions are pretty straightforward (similar to Lab 1), as for the button you want to configure the pin with the internal pullup and enable interrupts. Check the slides and examples about interrupts. Which event should trigger the interrupt? Level or edge? And should it be positive or negative?

Make sure you configure the interrupt and associate it with the ISR (see code examples on interrupts for reference). Something like:

```
void app_main()
{
  // init LEDs and button pins
  initLEDPin(LED1pin);
  initLEDPin(LED2pin);
  initButtonPin(Btnpin);

  // activate the interrupts for the GPIOs

  // add the ISR handler for the given pin

}
```

Then you program an ISR that handles the press of the button. Don’t forget to add the debouncing logic!

```
static void buttonPress_handler(void *arg)
{
  // disable interrupts for that pin


  // if enough time has passed since the button was last pressed
  // go to the next note


  // reactivate interrupts for that pin
}
```

To test this, instead of going to the next note, you could simply light up an LED and have an infinite loop inside the main. Whenever you press the button you should see the LED being switched ON or OFF.

Putting everything together

The device should:

1) Signal the user that it's time to start. Flash both LEDs for that.
2) For each string to be tuned (there are 6):
    a. Select the target frequency starting from the thinnest string (see table at the beginning)
    b. Sample sound for some time (1 second?)
    c. Print the note on the serial line
    d. If the detected frequency is lower than the target frequency, light up one LED
    e. If the detected frequency is lower than the target frequency, light up another LED
    f. If the frequency is close to the target one (say, plus/minus 3 Hz), then light up both LEDs.
3) If the user presses the button, the device moves to the next note in the list, corresponding to the next string.
4) Once all strings have been tuned, you can start over, or simply stop.

When you have finished, you may notice that frequency detection is not very precise, especially on the low tones. This is likely because our frequency estimation algorithm is very simple and the sound at lower tones doesn't come up at full power out of the speakers.

Don't bother too much if it doesn't work on low frequencies, as long as you can see it working on the higher ones, the lab is passed!

Otherwise, just make up your own tuning scale, or maybe try with a Ukulele one!

If you have time

As usual, keep your code clean and tidy and well-documented. In addition, there are a few improvements you can bring to the device. I'll give you some ideas:

1) Using hardware timers instead of software timers is a plus if done correctly.
2) You can implement a sort of "debounce" when counting the crossing. For example, if the maximum frequency you allow is 1kHz, then you should not expect two crossings to happen in a time that is less than about 0.5ms. In other words, you can discard every crossing that is "too quick". This removes a bit of the noise.
3) Play the target note on a speaker (you can use the DAC for that).