Advanced Computer Systems – 2017

Submit: Adar Zeitak and Yuval Lewi

Date: 29/03/17

Detecting whistles using the cc2650

Table of Contents

Introduction

Our goal at this project is to use the cc2650 Launchpad, a microphone and a buzzer to detect whistles and alarm accordingly. An application is to use the cc2650 as a key chain. Once a person loses his keys, he can whistle and the alarm will guide him to the right direction.

The Program

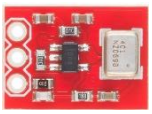
Program outline

The first stage is the sound recording. For this part, the microphone sampled data at a rate of 22100 Hz for 512 samples. After that, we used in-place FFT to understand weather the sound is a whistle or not. To lower the power consumption, the CPU then goes to IDLE for 800 ms, and then back to the first stage. If at any point the CPU detects whistling it starts a fast sampling – meaning without delay. After 10 straight whistles are detected[1], it alerts the user by activating a buzzer for 5 seconds. For a successful detection, a person needs to whistle for one second. This is the worst case, it could change based on the last time the CPU went to idle mode.

The program is built out of a main Task that waits until the ADC callback will wake it up (using a semaphore). Then it runs the logic, setting the alarm if needed, waits and calls for a microphone sampling.

[1] Actually it checks for 10 out of 12 for better performance

Sampling using a microphone



We used a microphone(ADMP401) to detect sound. This microphone uses an amplifier.

The samples move from the ADC to the RAM directly – meaning the CPU was on standby so we could save battery usage. The samples are saved in an array of the voltage got from the microphone. We save those numbers as integers and feed them to the Fourier transform. The microphone uses three legs – 3.3v, ground and DIO23 to transfer data.

Developing the whistle-detection algorithm

A Fourier transform converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa[2]. In this project, we computed the Fourier transform of the sampled audio signal and used heuristics to determine whether somebody whistles at the moment. To compute the Fourier transform, we adapted the Fast Fourier Transform (FFT) algorithm to run on the CC2650.

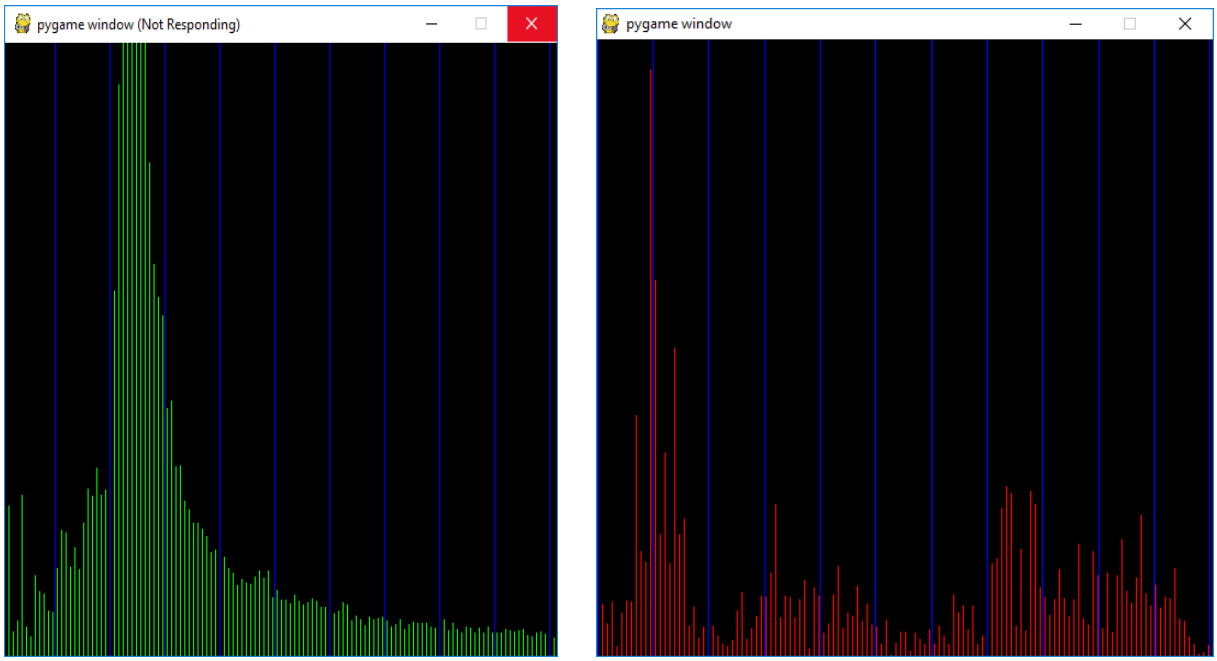


Figure 1 - FFT of a whistle sample

As a first step, we wrote python program to plot the FFT output and experiment with various detection heuristics (The program itself was added at the end as an attachment). This allowed us to develop and debug our detection algorithm easily before implementing it in C. As we played with the program, we saw that the spectrum of whistles has a unique structure. Looking at the FFT output as a graph of the magnitude of each frequency, we discovered two distinctive properties of whistle sounds (see Figures 1, 2):

1. Most of the energy was concentrated in frequencies between 1000 and 3000 Hz.
2. The whistle spectrum had a single peak, while other sounds tend to have a few separate peaks (harmonies).

Based on these properties, we devised the following algorithm to detect whistle sounds:

[2] Wikipedia Fast Fourier Transform - https://en.wikipedia.org/wiki/Fast_Fourier_transform

1. Compute the magnitude of each frequency, by squaring each element in the FFT output.
2. *avg* = average magnitude.
3. *whistle_avg* = average magnitude in the “whistle range” (1000-3000 Hz).
4. If *whistle_avg* < 2**avg*, return **False**.
5. Compute *filtered_spectrum,* by zeroing any element whose value is lower than 1/16 the maximal value.
6. *filt_sum* = Sum of *filtered_spectrum*.
7. Divide *filtered_spectrum* to stretches of nonzero elements and sum each.
8. If the sum of the stretch containing the maximal value is lower than 0.9**filt_sum*, return **False.**
9. Return **True**.

The algorithm tests property (1) in step (4) and property (2) in step (8). The computation of *filtered_spectrum* is required to cope with background noises. During testing we discovered that background noises create many weak peaks in the spectrum, making any sample violate the single-peak property. After adding the noise filtering and testing the algorithm against common non-whistle sounds we decided that it is robust enough and proceeded to implement it.

Converting our Python code to C code that will run on the CC2650 involved a few challenges. First, the CC2650 has very little RAM available, so storing both the sample buffer and the FFT result was impossible. As a solution, we used the FFT in-place variant, requiring no additional memory beyond the input buffer and leaving the result in the same buffer.

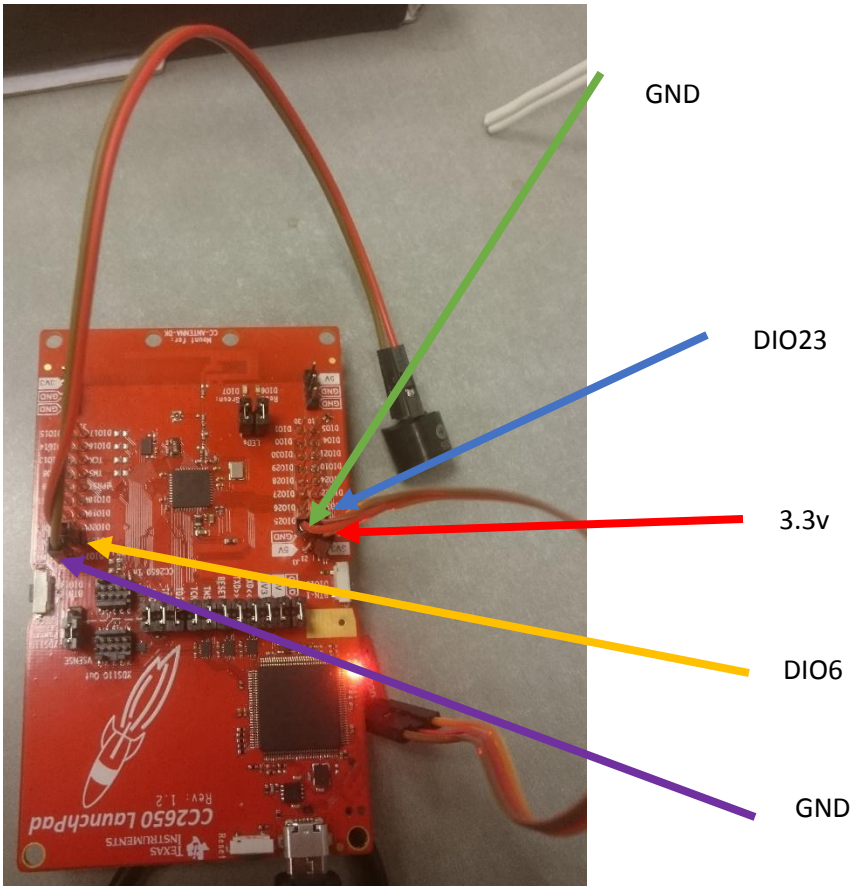
A second challenge was performing the floating-point arithmetic required for FFT. The CC2650 has a Cortex-M3 processor, which lacks a floating-point unit, so all floating-point computations are done using much slower software routines. As using the soft-float routines was problematic for us, we adapted our FFT code to use fixed-point numbers instead, and compute sines through table lookups. This resulted in faster code without depending on hardware support for floats.

Setting the alarm

To alert the user, we used TMB-05. It was connected to pin DIO6, which is also connected to the red LED on the board. When the program detects a whistle, it sets the pin to 3.3v for 5 seconds to turn on both the buzzer and the LED.

Results

The board



Duty cycle

We let the program run for 2 minutes while still printing. At this time the program detected three whistles. This result got a duty cycle of 5%. After about 10 minutes the duty cycle of 3%.

Attachments

The python code we used to find our heuristics for whistling detection.

The code uses numpy, pygame and pyaudio as external packages.

```
1.  import pyaudio
2.  import wave
3.  import numpy
4.  import pygame
5.  import struct
6.
7.  # given a frequency in Hz, compute which cell of the fft result
8.  # will contain it
9.  def freq_bin(freq):
10.     chunk_duration = float(CHUNK) / RATE
11.     cycles_per_chunk = freq * chunk_duration
12.     return int(cycles_per_chunk)
13.
```

```
# given an array with "islands" of consecutive nonzero values
# separated by spans of consecutive zeroes, compute the fraction
# of the mass contained in the island at <pos>
def piece_fraction(spector, pos):
    window_start = pos
    window_end = pos
    while window_start > 0 and spector[window_start] > 0.00001:
        window_start -= 1
    while window_end < len(spector)-1 and spector[window_end] > 0.000001:
        window_end += 1
    if sum(spector) < 0.000001:
        return 0
    return sum(spector[window_start:window_end]) / sum(spector)

# filter (make zero) everything lower than 1/16 the maximal value
def filter_weaks(spector):
    percent = max(spector)/16
    return [x if x > percent else 0 for x in spector]

def avg(values):
    return sum(values) / float(len(values))

def is_whistle(spector):
    avg_min = freq_bin(200)
    avg_max = freq_bin(10000)
    whistle_min = freq_bin(1000)
    whistle_max = freq_bin(3000)
    avg_area = map(lambda x:x**2,abs(spector[avg_min:avg_max]))
    whistle_area = map(lambda x:x**2,abs(spector[whistle_min:whistle_max]))
    filt_avg_area = filter_weaks(avg_area)
    peak = numpy.argmax(filt_avg_area)
    paramA = avg(whistle_area) > 2*avg(avg_area)
    paramB = piece_fraction(filt_avg_area, peak) > 0.9
    return (paramA and paramB, piece_fraction(filt_avg_area, peak), avg(whistle_area) /
 avg(avg_area))

# how many of the least-significant bits of each sample to remove.
# used to simulate low-resolution samples
LOST_BITS = 7
CHUNK = 512  # FFT block size
RATE = 22100

CHANNELS = 1
FORMAT = pyaudio.paInt16

SCREEN_SIZE = (512, 512)

screen = pygame.display.set_mode(SCREEN_SIZE)

p = pyaudio.PyAudio()

stream = p.open(format=FORMAT,
                channels=CHANNELS,
                rate=RATE,

                input=True,
                frames_per_buffer=CHUNK)

# a mask of all-ones except for LOST_BITS least significant bits
bitmask = ((-1) >> LOST_BITS) << LOST_BITS

```

```
try:
    run = True
    pause_frames = 0
    while True:
        pause_frames = max(0, pause_frames - 1)
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False
        if not run:
            break
        data = stream.read(CHUNK)
        data = struct.unpack('<%dh'%(CHUNK), data)
        data = [x & bitmask for x in data]

        spector = numpy.fft.rfft(data)/2
        whistle_detected,p1,p2 = is_whistle(spector)
        if whistle_detected:
            print "   YES  %.2f %.2f"%(p1,p2)
        else:
            print "NO      %.2f %.2f"%(p1,p2)
        if pause_frames > 0:
            continue
        if whistle_detected:
            pause_frames = 40
            color = (255,255,0)
        else:
                    color = (255,0,0)
                screen.fill((0,0,0))
                for i in xrange(0,len(spector),4):
                    pygame.draw.line(screen, color, (i,SCREEN_SIZE[1]), (i, SCREEN_SIZE[
    1]-int(abs(spector[i])/65536. * SCREEN_SIZE[1])))
                for i in xrange(0, 15000, 1000):
                    x = freq_bin(i)
                    pygame.draw.line(screen, (0,0,255), (x,SCREEN_SIZE[1]), (x, 0))
                pygame.display.flip()
       finally:
            pygame.quit()
            stream.stop_stream()
            stream.close()
            p.terminate()
```