Smart Dog Bowl Using FFT Analysis of Dog's Bark

Brian Kubisiak
brian.kubisiak@gmail.com
MSC 606

1 Motivation

In designing a smart dog bowl to open only for a specific dog, it is important that method used for identifying the dog be very accurate; there should never be a case where a dog is unable to access his food. Obvious approaches to this problem would be to put some sort of identification tag on the dog's collar, such as RFID or a modulated IR beacon.

However, these approaches can easily break down under fairly common circumstances—IR must have a clear path to the sensor, and RFID depends on a specific orientation. In addition, these approaches require the dog to be wearing his collar, and—in the case of IR—the collar must be powered. I believe that the most important part of this bowl is that a dog must never approach his bowl, only to have it not open due to something blocking the identification on his collar.

2 Method of Approach

To solve the problems described above, I have elected to identify the dog using its bark. Just as humans each have a distinct-sounding voice, a dog's bark contains a unique spectrum of frequencies. By analyzing these frequencies, I believe that a system can be designed that will open for one dog's bark but not another's.

The dog can never really "lose" his bark, so it is ideal in that the dog will always be able to open the dish on his own. Additionally, in the case that an error causes the dog to be misidentified, the dog will likely try again, reducing the probability of error.

One disadvantage of this design is that the dog will have to be trained to bark in order to open his food dish. However, this seems fairly easy, as teaching a dog to "speak" is a fairly common trick. Another obvious disadvantage is that the analysis will not get a distance measurement to the dog. In order to overcome this, I will use ultrasonic ranging sensors to determine whether or not the dog is within a 1 ft radius.

3 Operation

My design comprises a microphone with an audio amplifier, triggering logic for generating interrupts, an Arduino for performing the signal processing, and a servo motor for opening the dog

bowl.

All of the signal processing for this design will be performed on an Arduino board. The processor will sample the dog's bark, then perform an FFT on the data and compare it to save values. If the bark matches, the bowl will open.

There will be 4–6 ultrasonic proximity detectors to determine whether or not there is an object within 1 ft of the bowl. This will consist of an ultrasonic range finder and circuitry comparing it to a reference range. This detector will output a digital 1 if there is an object within a foot, and a 0 if there is not. Each range finder can detect in a 15° angle. By putting 4–6 of these in a circle around the bowl, they should be able to detect anything larger than a foot wide around the entire radius. More sensors can easily be used, at an increased cost.

The microphone will output a simple analog signal into the Arduino. In addition, some analog circuitry will be used to generate an interrupt whenever the analog input reaches a certain threshold in order to allow the Arduino to idle at lower power when data is not being read.

A block diagram for this design is shown in figure (1).

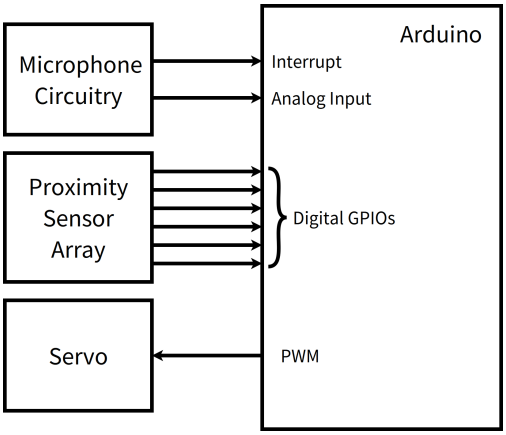

Figure 1: Top-level block diagram for the dog bowl.

3.1 Microphone Circuitry

All of the microphone circuitry is detailed in figure (2).

The microphone is powered from Vcc through a 2.2 k resistor, as recommended in the data sheet. The signal is then lowpass-filtered before being fed into the amplifier. The lowpass filter is designed with a corner frequency of 10 Hz using R2 = 100 k and C1 = 1 μF. This will filter out the DC component without affecting the AC signal.

The audio amplifier uses an op-amp with a single-sided supply in order to amplify the input signal. The amplifier is arranged in a noninverting configuration, with a total amplification factor of 560 (27.5 dB). The resistor Rf can be adjusted as needed to change the amplification. Note that the AC input may go below ground, but this will be clipped by the amplifier.

The output of the amplifier is then fed directly into the ADC on the Arduino board. This amplified signal is also fed into the comparator U2 in order to generate an interrupt to tell the Arduino to

begin recording the audio. The trigger level is set with R4 and R5. I have found that a level of around 1.7 V works well. This interrupt trigger is fed into a GPIO pin on the Arduino to generate an interrupt.

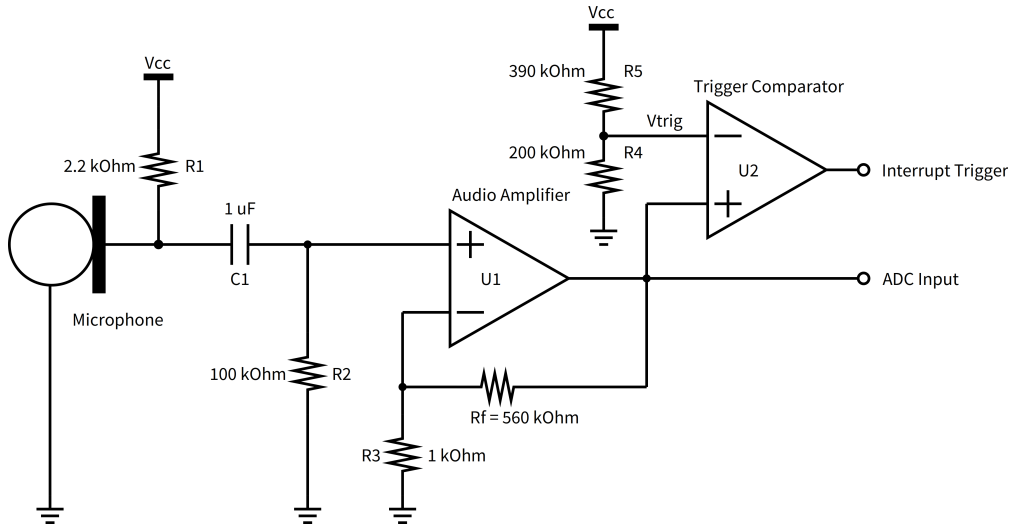

Figure 2: Microphone amplifier and trigger generator.

3.2 Proximity Sensor Array

The proximity sensor array is shown in figure (3).

The sensor array is fed by five ultrasonic range finders. Each range finder outputs an analog voltage representing the distance to the nearest object (the exact relation is unspecified in the data sheet; I will figure out the relation once the sensors arrive in the mail). The output of these sensors will be fed into comparators, comparing them to a fixed voltage. This voltage will correspond to approximately 1 ft from the ultrasonic sensors. The voltage is created using a voltage divider with the resistors R6 and R7.

The outputs from all of these comparators will go into GPIO pins on the Arduino. When the microphone circuit triggers a recording, the Arduino will first check that at least one of the proximity sensors is triggered before computing the FFT of the recording.

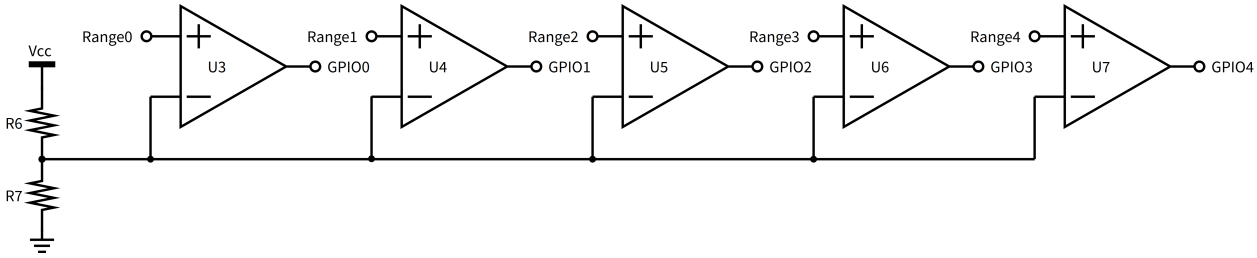

Figure 3: Circuit for the array of proximity sensors.

3.3 Servo Motor

The servo motor is a standard Parallax servo motor with 180° of motion. The position of the servo is controlled through PWM. One of the PWM outputs on the Arduino will switch the servo between the open position and the closed position when needed.

4 Source Code Listing

Note that the following code is still a work in progress.

4.1 FFT Function Declarations

```
/*
 * fft.h
 *
 * Perform Fast Fourier Transforms on data.
 *
 * This code is used for computing a low-footprint, high-speed FFT on an array
 * of data. Because of the limitations of the platform that I am working on,
 * several optimizations have to be made in order for the code to work.
 * Specifically, the limitations I have are:
 *
 *      8 kB of RAM
 *      No floating-point unit
 *      8 -> 16 bit multiplier
 *      8 bit adder (for speed)
 *
 * Some limitations are self-imposed in order to produce faster code. Each data
 * point is a complex number with 8 bits for the real part and 8 bits for the
 * imaginary.
 *
 * Revision History:
 *      16 Apr 2015     Brian Kubisiak      Initial revision.
 */


#ifndef _FFT_H_
#define _FFT_H_


#include "data.h"   /* Complex data type, size of array, etc. */


/*
 * fft
 *
 * Description: Computes the fast Fourier transform (FFT) of an array of input
 *              data. This implementation is in-place, so it will take O(1)
 *              space. Additionally, optimizations are performed assuming that
 *              the data to transform is purely real. The implementation assumes
 *              8-bit characters and 16-bit shorts. It attempts to avoid 16-bit
 *              additions/multiplications wherever possible in order to cut down
 *              on the number of clocks.
```

```
 *
 * Arguments:   data An array of complex numbers for performing the FFT. This
 *                   FFT assumes that the array contains 'SAMPLE_SIZE' values of
 *                   type 'complex'. Both of these are defined in 'data.h'.
 *
 * Limitations: Assumes that the input data is purely real. Because there is no
 *              FPU and fixed-point arithmetic is used, the resulting FFT will
 *              not be normalized to anything sensible.
 */
void fft(complex *data);


#endif /* end of include guard: _FFT_H_ */
```

4.2 FFT Code

```
/*
 * fft.c
 *
 * Perform Fast Fourier Transforms on data.
 *
 * This code is used for computing a low-footprint, high-speed FFT on an array
 * of data. Because of the limitations of the platform that I am working on,
 * several optimizations have to be made in order for the code to work.
 * Specifically, the limitations I have are:
 *
 *      8 kB of RAM
 *      No floating-point unit
 *      8 -> 16 bit multiplier
 *      8 bit adder (for speed)
 *
 * Some limitations are self-imposed in order to produce faster code. Each data
 * point is a complex number with 8 bits for the real part and 8 bits for the
 * imaginary.
 *
 * Revision History:
 *      16 Apr 2015     Brian Kubisiak      Initial revision.
 */


#include "fft.h"

extern complex root[SAMPLE_SIZE];


/*
 * fft
 *
 * Description: Computes the fast Fourier transform (FFT) of an array of input
 *              data. This implementation is in-place, so it will take O(1)
 *              space. Additionally, optimizations are performed assuming that
 *              the data to transform is purely real. The implementation assumes
 *              8-bit characters and 16-bit shorts. It attempts to avoid 16-bit
 *              additions/multiplications wherever possible in order to cut down
 *              on the number of clocks.
 *
```

```
 * Arguments:   data An array of complex numbers for performing the FFT. This
 *                   FFT assumes that the array contains 'SAMPLE_SIZE' values of
 *                   type 'complex'. Both of these are defined in 'data.h'.
 *
 * Limitations: Assumes that the input data is purely real. Because there is no
 *              FPU and fixed-point arithmetic is used, the resulting FFT will
 *              not be normalized to anything sensible.
 *
 * Notes:       A lot of the notation below is made up. Basically, we do log2(N)
 *              passes over the data, where each pass will iterate over a
 *              cluster of butterflies (try googling 'FFT butterfly' if you
 *              don't know what this is). Each cluster of butterflies is
 *              separated by a certain stride value, so we can just continue to
 *              increment by this stride until the data is covered. Then, start
 *              the next pass.
 *
 *              If you don't understand how this works, try staring at FFT
 *              butterflies a bit longer and it will hopefully make sense. If
 *              that doesn't work, try eating ice cream because yum ice cream.
 */

void fft(complex *data)
{
    /*
     * The stride of each pass over the data is a measure of the distance
     * between the two data points combined together in a butterfly. It is
     * always a power of two.
     */
    unsigned int stride;
    unsigned char i, j, k;      /* Loop indices. */


    /* We start off with two separate clusters of butterflie nodes filling the
     * entire data set. */
    stride = SAMPLE_SIZE / 2;

    /* We need to perform log2(N) iterations over the data in order to fully
     * transform it. */
    for (i = 0; i < LOG2_SAMPLE_SIZE; i++)
    {
        /*
         * Keep striding through the data until we cover all of it. Note that we
         * only go to 'SAMPLE_SIZE / 2', since each butterfly covers 2 data
         * points.
         */
        for (j = 0; j < SAMPLE_SIZE / 2; j += stride + 1)
        {
            /*
             * Iterate over every butterfly in the cluster. This will use one
             * data point in the cluster, and one in the next cluster. We then
             * stride over the next butterfly cluster to avoid redoing this
             * computation.
             */
            for (k = j; k < j + stride; k++)
            {
                /* Get the two data points that we are transforming. */
                complex a = data[k];
                complex b = data[k+stride];

```

```
100                 /* Now transform them using a butterfly. */
101                 data[k]         = add(a, mul(b, root[k]));
102                 data[k+stride]  = add(a, mul(b, root[k+LOG2_SAMPLE_SIZE]));
103             }
104         }
105
106         /* Reduce the stride for the next pass over the data. */
107         stride /= 2;
108     }
109 }
```

4.3 Complex Number Data Type

```
1  /*
2   * data.h
3   *
4   * Data types and constants for analyzing a frequency spectrum.
5   *
6   * This file describes the complex number data type, as well as functions for
7   * adding and multiplying complex numbers. Constants for determining the number
8   * of samples to use are also defined here.
9   *
10  * Revision History:
11  *      16 Apr 2015     Brian Kubisiak      Initial revision.
12  */
13
14 #ifndef _DATA_H_
15 #define _DATA_H_
16
17
18 #define SAMPLE_SIZE         1024    /* Number of samples in the buffer. */
19 #define LOG2_SAMPLE_SIZE    10      /* Base two logarithm of the sample size. */
20
21
22 /*
23  * complex
24  *
25  * Description: Data type for holding a complex number. This data type uses
26  *              Cartesian coordinates for holding the number, so it is optimized
27  *              for addition rather than multiplication.
28  *
29  * Members:     real  The real part of the complex number.
30  *              imag  The imaginary part of the complex number.
31  */
32 typedef struct _complex {
33     char real;
34     char imag;
35 } complex;
36
37
38 /*
39  * add
40  *
41  * Description: Adds together two complex numbers in the Cartesian plane.
42  *
43  * Arguments:   a  The first number to add.
```

```
 *              b  The second number to add.
 *
 * Returns:     Returns the complex number that is the sum of the two inputs.
 */
complex add(complex a, complex b);


/*
 * mul
 *
 * Description: Computes the product of two complex numbers in the Cartesian
 *              plane.
 *
 * Arguments:   a  First number to multiply.
 *              b  Second number to multiply.
 *
 * Returns:     Returns the complex number that is the product of the two
 *              inputs.
 *
 * Notes:       The number is stored in rectangular coordinates, so taking the
 *              product will be a bit slow.
 */
complex mul(complex a, complex b);


#endif /* end of include guard: _DATA_H_ */
```

4.4 Complex Number Operations

```
/*
 * data.c
 *
 * Data types and constants for analyzing a frequency spectrum.
 *
 * This file describes the complex number data type, as well as functions for
 * adding and multiplying complex numbers. Constants for determining the number
 * of samples to use are also defined here.
 *
 * Revision History:
 *      16 Apr 2015     Brian Kubisiak      Initial revision.
 */

#include "data.h"



/*
 * add
 *
 * Description: Adds together two complex numbers in the Cartesian plane.
 *
 * Arguments:   a  The first number to add.
 *              b  The second number to add.
 *
 * Returns:     Returns the complex number that is the sum of the two inputs.
```

```
 */
complex add(complex a, complex b)
{
    complex res;

    /* Real part of result is sum of real parts of inputs. */
    res.real = a.real + b.real;
    /* Imaginary part of result is sum of imaginary parts of inputs. */
    res.imag = a.imag + b.imag;

    return res;
}


/*
 * mul
 *
 * Description: Computes the product of two complex numbers in the Cartesian
 *              plane.
 *
 * Arguments:   a  First number to multiply.
 *              b  Second number to multiply.
 *
 * Returns:     Returns the complex number that is the product of the two
 *              inputs.
 *
 * Notes:       The number is stored in rectangular coordinates, so taking the
 *              product will be a bit slow.
 */
complex mul(complex a, complex b)
{
    complex res;

    /* (a + jb)(c + jd) = ac + jad + jbc + j^2 cd = (ac - cd) + j(ad + bc) */

    /* Real part of result comes from product of real parts minus product of
     * imaginary parts. */
    res.real = (a.real * b.real) - (a.imag * b.imag);
    /* Imaginary part is sum of cross products of inputs. */
    res.imag = (a.real * b.imag) + (a.imag * b.real);

    return res;
}
```