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 Hardware Operation

My design comprises a microphone with an audio amplifier, triggering logic for generating interrupts, ultrasonic rangefinders for measuring proximity, 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 saved values. If the bark matches, the bowl will open.

There will be 4 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 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 a slightly 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).

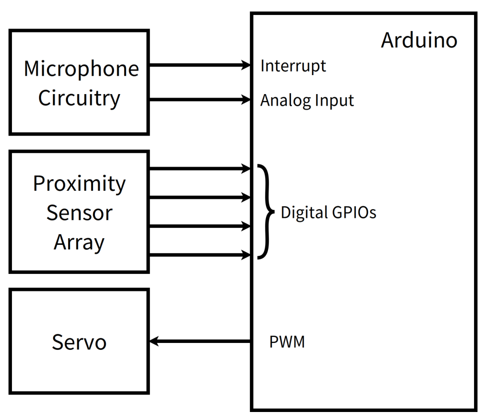

Figure 1: Top-level block diagram for the dog bowl.

3.1 Microphone Circuitry

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

The microphone is powered from Vcc through a 2.2 kΩ 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 kΩ 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.

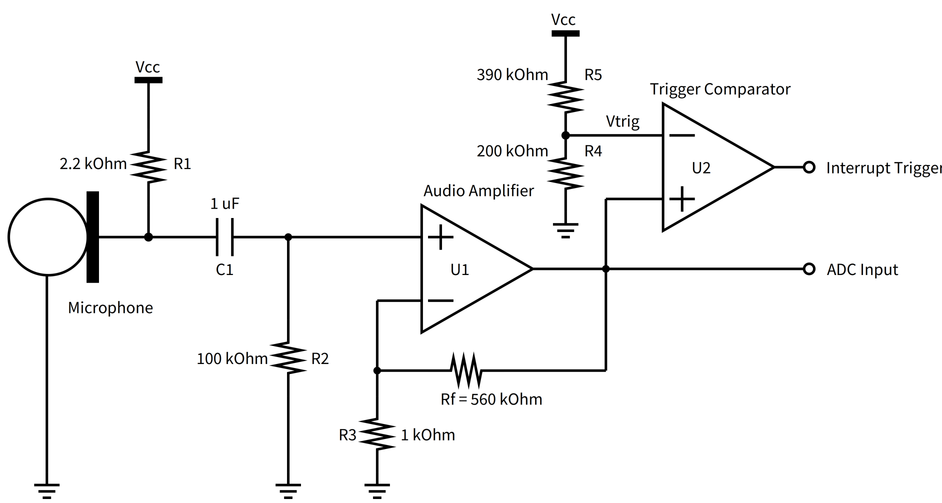

Figure 2: Microphone amplifier and trigger generator.

3.2 Proximity Sensor Array

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

The sensor array is fed by ultrasonic range finders. Each range finder outputs an analog voltage representing the distance to the nearest object (the exact relation is not important to this design). 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. I have found that a voltage level of 0.76 V works well for identifying objects approximately 1 ft away.

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.

Note that the comparators used in my design are open drain, so they do not drive the output to Vcc; rather, they put the output in a high-impedance state that requires a pull-up resistor. The Arduino GPIO pins have internal pull-up resistors, so these are not included in my analog circuitry.

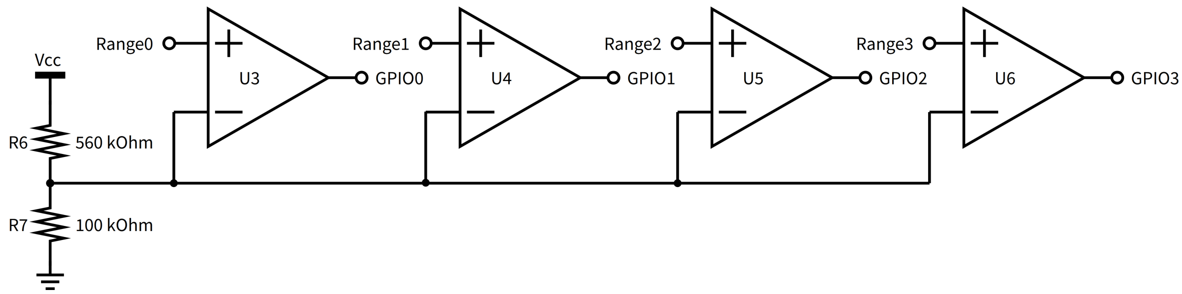

Figure 3: Circuit for the array of proximity sensors.

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 Software Operation

The main loop for the software is a simple finite state machine, shown in figure (4). The machine starts in the INIT state. Once data is available to analyze, the program checks if there is anything in the proximity of the bowl. If there is, then the FFT analysis is performed. If not, the data collection is reset.

If the FFT analysis finds that the frequency spectrum matches the spectrum of the key, then the bowl is opened. Otherwise, the data collection is reset. Once the bowl is open, it will just wait until nothing is nearby (i.e. the proximity sensors are no longer activated). At this point, the bowl will be reset.

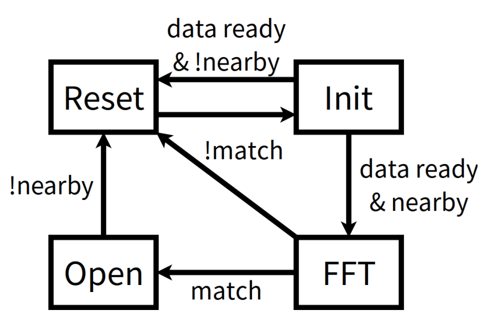

Figure 4: Finite state machine for controlling the dogbowl.

5 Results

The end result of my design is moderately successful. The Fourier analysis will usually trigger after a few seconds of barking, and is unlikely to get a false positive. This project is a good proof-of-concept, and I believe with a few improvements, it would work fairly reliably.

The first improvement I would make would be to use a DSP for the signal processing; with a more powerful chip, my project could perform a larger FFT and more sophisticated error analysis. Additionally, adding a faster ADC would allow me to sample the full audible frequency spectrum. This would likely reduce the false positives by comparing a wider spectrum of frequencies.

Another limitation I encountered is that my microphone is poorly placed in the final design. When I was constructing the circuit, I did not consider how the whole design would be assembled and packaged. As a result, the microphone ended up in a place that did not get a clear audio signal. The most important lesson that I have learned from this project is to always think about the final packaging and design so that choices made at the beginning do not limit the design later.

The last problem I had with my design is that the ultrasonic sensors should be angled upwards at a steeper angle. As they are now, they can be triggered by the floor. This is an issue because the bowl will remain open even after the dog has left. Again, anticipating the final design would have allowed me to fix this.

6 Source Code Listing

6.1 Analog Data Collection Function Declarations

```
/*
 * adc.h
 *
 * Functions for collecting data over the ADC.
 *
 * This file contains functions that handle data collection from the ADC using
 * interrupts. The ADC will be run at set intervals and generate an interrupt
 * every time it has a new data point. Functions in this file will record the
 * data point and disable further interrupts once the buffer is full. Once the
 * buffer is no longer in use, interrupts can be re-enabled when needed.
 *
 * Peripherals Used:
 *      ADC
 *      External interrupts
 *
 * Pins Used:
 *      PF0
 *      PD0
 *
 * Revision History:
 *      05 Jun 2015     Brian Kubisiak      Initial revision.
```

```
 *      06 Jun 2015     Brian Kubisiak      Added external trigger.
 */

#ifndef _ADC_H_
#define _ADC_H_


#include "data.h"

/*
 * init_adc
 *
 * Description: This function initializes the ADC peripheral so that the proper
 *              pins are allocated for use. After this function, the ADC will
 *              *not* be running; the 'adc_start_collection' function should be
 *              called before data will be collected. This initialization
 *              involves:
 *               - Writing to ADMUX and ADCSRB to select the input channel.
 *               - Enable ADC by writing to ADCSRA.
 *               - Left-adjust the data input by writing to ADMUX.
 *               - Set the trigger source using ADCSRB.
 *               - Setting up interrupts for autotriggering.
 *               - Setting up external interrupt.
 *
 * Notes:       This function will initialize the ADC to use PF0. If this pin is
 *              used for another purpose, these functions will not work
 *              properly.
 */
void init_adc(void);


/*
 * adc_get_buffer
 *
 * Description: Get a pointer to the buffer of data that has been filled by the
 *              ADC. Note that this buffer should be retrieved each time that
 *              the data collection completes and should not be used globally.
 *
 * Returns:     Returns a pointer to the buffer containing the data collected by
 *              the ADC.
 *
 * Notes:       This kind of makes the buffer into a global variable, so be
 *              careful where this is used. Also note that the buffer should
 *              only be used once it is filled.
 */
complex *adc_get_buffer(void);

/*
 * is_data_collected
 *
 * Description: Determines whether or not the data has been fully collected. If
 *              the buffer is full of new data, this function returns nonzero.
 *              If the ADC is still collecting data for the buffer, returns 0.
 *              This is used to find if the data is ready to run through the
 *              FFT.
 *
```

```
 * Returns:     If the buffer is full of new data, return nonzero. Else, return
 *              zero.
 *
 * Notes:       The data collection is reset when the 'adc_start_collection'
 *              function is called.
 */
unsigned char is_data_collected(void);

/*
 * adc_reset_buffer
 *
 * Description: Resets the data buffer that is filled by the ADC. This function
 *              will clear empty the buffer and cause the ADC to begin filling
 *              from the beginning. Note that this function does not start the
 *              ADC data collection; the buffer will be refilled from the
 *              beginning once the 'adc_start_collection' function is called.
 *
 * Notes:       This function should not be called while the buffer is in the
 *              process of being filled. This may cause unexpected race
 *              conditions.
 */
void adc_reset_buffer(void);


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

6.2 Analog Data Collection Functions

```
/*
 * adc.c
 *
 * Functions for collecting data over the ADC.
 *
 * This file contains functions that handle data collection from the ADC using
 * interrupts. The ADC will be run at set intervals and generate an interrupt
 * every time it has a new data point. Functions in this file will record the
 * data point and disable further interrupts once the buffer is full. Once the
 * buffer is no longer in use, interrupts can be re-enabled when needed.
 *
 * Peripherals Used:
 *      ADC
 *      External interrupts
 *
 * Pins Used:
 *      PF0
 *      PD0
 *
 * Revision History:
 *      05 Jun 2015     Brian Kubisiak      Initial revision.
 *      06 Jun 2015     Brian Kubisiak      Added external trigger.
 *      08 Jun 2015     Brian Kubisiak      Added pullup resistor to INT0.
 */

#include <avr/io.h>
```

```
#include <avr/interrupt.h>

#include "adc.h"

/* Initial values for the ADC configuration registers. */
#define ADMUX_VAL   0x60
#define ADCSRA_VAL  0x8F
#define ADCSRB_VAL  0x00
#define DIDR0_VAL   0x01
#define DIDR2_VAL   0x00

/* Initial values for external interrupt configuration. */
#define EICRA_VAL   0x03
#define EIMSK_VAL   0x01

/* Add pullup resistor to interrupt pin. */
#define PORTD_VAL   0xFF

/* ORing this with ADCSRA will begin the data collection process. */
#define ADCSTART    0x60

static complex databuf[SAMPLE_SIZE];
static unsigned int bufidx = 0;
static unsigned char buffull = 0;
static unsigned char collecting = 0;

/*
 * adc_start_collection
 *
 * Description: Start collecting a new buffer of data. This function will enable
 *              auto-triggering off of the ADC interrupt and start a new
 *              conversion to start the chain of data collection. Once the
 *              buffer is full, the interrupt vector will disable the
 *              autotriggering automatically.
 *
 * Notes:       This function should not be called until the buffer is filled
 *              and data collection halts. This can be checked with the
 *              'is_data_collected' function.
 */
static void adc_start_collection(void)
{
    /* Reset the index to load values into the start of the buffer. */
    bufidx = 0;

    /* Buffer is no longer full. */
    buffull = 0;

    /* Set the flag signaling that collection is in progress. */
    collecting = 1;

    /* Enable autotriggering and start the first conversion. */
    ADCSRA |= ADCSTART;
}

/*
 * init_adc
```

```
 *
 * Description: This function initializes the ADC peripheral so that the proper
 *              pins are allocated for use. After this function, the ADC will
 *              *not* be running; the 'adc_start_collection' function should be
 *              called before data will be collected. This initialization
 *              involves:
 *               - Writing to ADMUX and ADCSRB to select the input channel.
 *               - Enable ADC by writing to ADCSRA.
 *               - Left-adjust the data input by writing to ADMUX.
 *               - Set the trigger source using ADCSRB.
 *               - Enable the ADC interrupt.
 *               - Setting up interrupts for autotriggering.
 *               - Setting up external interrupt.
 *
 * Notes:       This function will initialize the ADC and external interrupt to
 *              use PF0 and PD0. If these pins are used for another purpose,
 *              these functions will not work properly.
 */
void init_adc(void)
{
    /* Set all the configuration registers to their initial values. */
    ADMUX   = ADMUX_VAL;
    ADCSRA  = ADCSRA_VAL;
    ADCSRB  = ADCSRB_VAL;
    DIDR0   = DIDR0_VAL;
    DIDR2   = DIDR2_VAL;

    /* Reset the buffer collection. */
    adc_reset_buffer();

    /* Add pullup resistor to the INT0 pin. */
    PORTD = PORTD_VAL;

    /* Activate the external interrupt for triggering a recording. */
    EICRA = EICRA_VAL;
    EIMSK = EIMSK_VAL;
}

/*
 * adc_get_buffer
 *
 * Description: Get a pointer to the buffer of data that has been filled by the
 *              ADC. Note that this buffer should be retrieved each time that
 *              the data collection completes and should not be used globally.
 *
 * Returns:     Returns a pointer to the buffer containing the data collected by
 *              the ADC.
 *
 * Notes:       This kind of makes the buffer into a global variable, so be
 *              careful where this is used.
 */
complex *adc_get_buffer(void)
{
    /* Return the current buffer. */
    return databuf;
}
```

```

/*
 * is_data_collected
 *
 * Description: Determines whether or not the data has been fully collected. If
 *              the buffer is full of new data, this function returns nonzero.
 *              If the ADC is still collecting data for the buffer, returns 0.
 *              This is used to find if the data is ready to run through the
 *              FFT.
 *
 * Returns:     If the buffer is full of new data, return nonzero. Else, return
 *              zero.
 *
 * Notes:       The data collection is reset when the 'adc_start_collection'
 *              function is called.
 */
unsigned char is_data_collected(void)
{
    /* Data is collected once the buffer is full. */
    return buffull;
}


/*
 * adc_reset_buffer
 *
 * Description: Resets the data buffer that is filled by the ADC. This function
 *              will clear empty the buffer and cause the ADC to begin filling
 *              from the beginning. Note that this function does not start the
 *              ADC data collection; the buffer will be refilled from the
 *              beginning once the 'adc_start_collection' function is called.
 *
 * Notes:       This function should not be called while the buffer is in the
 *              process of being filled. This may cause unexpected race
 *              conditions.
 */
void adc_reset_buffer(void)
{
    /* Start buffer collection from the beginning. */
    bufidx = 0;

    /* Buffer is no longer full. */
    buffull = 0;

    /* No longer collecting data. */
    collecting = 0;
}

/*
 * ADC_vect
 *
 * Description: Interrupt vector for the ADC interrupt. When this interrupt
 *              occurs, the function will store the new data point if the buffer
 *              is not yet full. Then, the function will check to see if the
 *              buffer is now full, updating the flag accordingly. Once the
 *              buffer is full, data collection is disabled.
```

```
 *
 * Notes:       The interrupt should be automatically reset in hardware.
 */
ISR(ADC_vect)
{
    /* If the buffer is not yet full, record the data. */
    if (!buffull)
    {
        /* Take the upper 8 bits of the ADC as the real part of the signal. The
         * imaginary part is zero. */
        databuf[bufidx].real = ADCH;
        databuf[bufidx].imag = 0;

        /* Next data point should be stored in the next slot. */
        bufidx++;

        /* Check to see if the buffer is full. */
        if (bufidx == SAMPLE_SIZE)
        {
            /* When full, set the flag. */
            buffull = 1;

            /* Disable further data collection. */
            ADCSRA = ADCSRA_VAL;
        }
    }
    /* If the buffer is already full, then we triggered once too many
     * conversions. This interrupt can be ignored. */

    /* Interrupt flag is automatically turned off in hardware. */
}

/*
 * INT0_vect
 *
 * Description: Triggers the ADC data collection on an external interrupt. Once
 *              the amplitude of the audio input goes above a certain level,
 *              this interrupt will fire and begin recording data. If data is
 *              already being collected, or the buffer is already full, then the
 *              interrupt will be ignored.
 *
 * Notes:       The interrupt should be automatically reset in hardware.
 */
ISR(INT0_vect)
{
    /* If we aren't already collecting, start the collection. */
    if (!collecting) {
        adc_start_collection();
    }
    /* Else, just ignore this interrupt. */

    /* The interrupt flag is cleared automatically in hardware. */
}
```

6.3 Complex Number Data Type

```
/*
 * data.h
 *
 * 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.
 *      04 Jun 2015     Brian Kubisiak      Changes to SAMPLE_SIZE macro.
 */

#ifndef _DATA_H_
#define _DATA_H_



#ifndef SAMPLE_SIZE
#define SAMPLE_SIZE         64      /* Number of samples in the buffer. */
#define LOG2_SAMPLE_SIZE    6
#endif


/*
 * complex
 *
 * Description: Data type for holding a complex number. This data type uses
 *              Cartesian coordinates for holding the number, so it is optimized
 *              for addition rather than multiplication.
 *
 * Members:     real  The real part of the complex number.
 *              imag  The imaginary part of the complex number.
 */
typedef struct _complex {
    char real;
    char imag;
} complex;


/*
 * 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);


```

```
/*
 * 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_ */
```

6.4 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;
}
```

6.5 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.
 *      06 Jun 2015     Brian Kubisiak      Added method for FFT comparison.
 */


#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);


/*
 * is_fft_match
 *
 * Description: Determines if the given frequency spectrum data is an
 *              approximate match for the previously recorded data. The data
 *              that this will be compared to is stored in ROM at compile time.
 *              This function will first take the (integer) log2 of the
 *              magnitude of the input data in order to normalize it. Then, the
 *              difference between this data and the comparison values is
 *              calculated, squared, and accumualted to get a measure of the
 *              error. This is compared to a set threshold: above the threshold,
 *              0 is returns; below the threshold, 1 is returned.
 *
 * Arguments:   data -- The data to compare to the previously-recorded data to
 *                      determine whether or not there is a match.
```

```
 *
 * Returns:     Returns 0 if the input data is dissimilar to the comparison
 *              data. Returns 1 if the input data matches the comparison data,
 *              within some error.
 *
 * Notes:       This function is very slow and probably won't give very good
 *              results. Ideally, some more sophisticated analysis on a more
 *              powerful chip should be used.
 */
unsigned char is_fft_match(complex *data);


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

6.6 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.
 *      06 Jun 2015     Brian Kubisiak      Added method for FFT comparison.
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "fft.h"

#define ERROR_THRESHOLD 30

extern complex root[SAMPLE_SIZE];       /* Roots of unity for the FFT. */
extern unsigned char key[SAMPLE_SIZE];  /* Spectrum that will open the bowl. */

/*
 * 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 int 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; j += 2*stride)
        {
            /*
```

```
             * 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];

                /* Since the roots are stored in bit-reversed order, we can just
                 * index into the array with the butterfly index. */
                complex w = root[j];

                /* The negative of the root is just 180 degrees around the unit
                 * circle. */
                complex neg_w = root[j + SAMPLE_SIZE / 2];

                /* Now transform them using a butterfly. */
                data[k]         = add(a, mul(b, w));
                data[k+stride]  = add(a, mul(b, neg_w));
            }
        }

        /* Reduce the stride for the next pass over the data. */
        stride /= 2;
    }
}


/*
 * is_fft_match
 *
 * Description: Determines if the given frequency spectrum data is an
 *              approximate match for the previously recorded data (the 'key').
 *              The key that this data will be compared to is stored in ROM at
 *              compile time. This function will first take the (integer) log2
 *              of the magnitude of the input data in order to normalize it.
 *              Then, the difference between this data and the key is
 *              calculated, squared, and accumulated to get a measure of the
 *              error. This is compared to a set threshold: above the threshold,
 *              0 is returns; below the threshold, 1 is returned.
 *
 * Arguments:   data -- The data to compare to the key to determine whether or
 *                      not there is a match.
 *
 * Returns:     Returns 0 if the input data is dissimilar to the key.
 *              Returns 1 if the input data matches the key, within some error.
 *
 * Notes:       This function is very slow and probably won't give very good
 *              results. Ideally, some more sophisticated analysis on a more
 *              powerful chip should be used.
 */
unsigned char is_fft_match(complex *data)
{
```

```
    double mag;
    unsigned int i;
    unsigned char cmpval;
    unsigned long err = 0;

    /* Transform each point of the input data. */
    for (i = 0; i < SAMPLE_SIZE; i++)
    {
        /* First, take the log of the magnitude of the data. This will fit in an
         * 8-bit char. */
        mag = data[i].real * data[i].real + data[i].imag * data[i].imag;
        cmpval = (char)log10(mag);

        /* Now calculate the absolute value of the error, and accumulate it. */
        err += abs(cmpval - key[i]);
    }

    /* Return true iff the error is below the error threshold. */
    return (err < ERROR_THRESHOLD);
}
```

6.7 Frequency Spectrum Key

```
/*
 * key.c
 *
 * Power spectrum for the key to the dog bowl.
 *
 * Contains the magnitude of the power spectrum of the dog bark that will unlock
 * the dog bowl. This spectrum is compared to the recorded spectrum in order to
 * identify the dog that barked. If the spectra match, the dog bowl will open.
 *
 * Revision History:
 *      06 Jun 2015     Brian Kubisiak      Initial revision.
 *      09 Jun 2015     Brian Kubisiak      Working key added.
 */

#include "data.h"

/* Frequency spectrum that unlocks the dog bowl, obtained empiracally. */
unsigned char key[SAMPLE_SIZE] = {
    2, 3, 3, 4, 4, 3, 4, 3, 4, 2, 4, 3, 4, 4, 4, 3, 4, 3, 4, 4, 4, 4, 3, 4, 4,
    3, 3, 3, 4, 4, 3, 4, 4, 2, 4, 4, 4, 4, 4, 3, 3, 3, 3, 4, 3, 4, 4, 4, 3, 4,
    3, 3, 4, 2, 3, 3, 4, 4, 3, 3, 3, 4, 4, 4,
};
```

6.8 Generating Roots of Unity

```
#!/usr/bin/env python2

import math         # Mathematical constants (e, pi)
```

```
import sys          # Command-line arguments
import datetime     # Generating revision dates


# Header string for including at the top of the generated .c file containing the
# roots of unity. Assumes that the output will go into a file called ’roots.c’.
header = ’’’
/*
 * roots.c
 *
 * Constants representing the roots of unity.
 *
 * This file contains an array of the nth roots of unity. The total number of
 * roots is determined by the constant ’SAMPLE_SIZE’, which should be defined in
 * the ’data.h’ header file (or at compile time in the Makefile).
 *
 * DO NOT MODIFY THIS FILE BY HAND. IT IS GENERATED AUTOMATICALLY BY THE
 * genroots.py PYTHON SCRIPT.
 *
 * Revision History:
 *      04 Jun 2015     Brian Kubisiak      Initial revision.
 *
 * Last Generated:
 *      %s
 */


#include "data.h"
’’’ % datetime.date.today().strftime("%d %b %Y")

dataheader = ’’’
/*
 * root
 *
 * Description: This array contains the nth roots of unity for calculating the
 *              FFT. The roots in this array contain an 8-bit real part and an
 *              8-bit imaginary part. The roots are organized in a modified
 *              bit-reversed order in order to make the accesses easier.
 *
 * Notes:       Due to the ordering of the roots, if the ith root is at
 *              ’root[j]’, then its negative is at ’root[j + SAMPLE_SIZE/2]’.
 */
const complex root[SAMPLE_SIZE] = {’’’

datafooter = ’’’};’’’

footer = ’’’
’’’

def genroots(n):
    """ This function takes in a number n and generates the nth roots of unity.
    The resulting roots are returned in an array.

    args:
      n -- The number of roots to generate.

```

```
    returns:
      Returns an array containing the nth roots of unity.
    """

    # Start with an empty array of roots.
    roots = []

    # First root. We can exponentiate this to get the other roots.
    w = math.e ** (2j * math.pi / n)

    # Need to generate all n roots.
    for i in range(n):
        roots.append(w ** i)

    return roots

def bitreverse(roots):
    """ This function takes in the nth roots of unity and rearranges them in
    bit-reversed order. Specifically, the root at index 0bwxyz will be moved to
    index 0bzyxw, assuming that the root is in the first half of the array.

    If the root is in the second half of the array, then it is just the negative
    of a root in the first half. The roots in the second half of the array will
    be ordered the same as their negatives in the first half. This will make it
    easier to find negations of roots.

    args:
      n -- number of roots of unity
      roots -- array containing all the nth roots of unity

    returns:
      Returns a new array of roots containing the values in the bit-reversed
      order described above.
    """

    # Begin by copying the roots array.
    out = roots

    # Get the number of roots from the length of the array
    n = len(roots)

    # Loop over the first half of the values. Note that the second half will be
    # rearranged in the same pattern as the first half.
    for i in range(n / 2):
        # First, compute a string representation of the number with the given
        # width.
        b = '{:0{width}b}'.format(i, width = int(math.log(n/2, 2)))
        # Now, reverse the string and convert it into an integer.
        j = int(b[::-1], 2)

        # Swap the values at these indices in both halves of the array.
        out[i], out[j] = out[j], out[i]
        out[i + n/2], out[j + n/2] = out[j + n/2], out[i + n/2]

    return out

```

```
def printroots(roots):
    """ This function prints out all the roots of unity in a format that can be
    included as a C header file. The output will create an array 'root' that
    contains the nth roots of unity of type 'complex' with fields 'real' and
    'imag'. Note that this does not comply with the standard complex data type
    in C.

    args:
      roots -- the roots of unity to print.
    """

    # Print the file header and some documentation
    print header
    print dataheader

    for i in roots:
        print "    { .real = %s, .imag = %s }," % (int(round(i.real * 127)),
                                                   int(round(i.imag * 127)))

    # Close the array and print some closing documentation.
    print datafooter
    print footer

def main():
    try:
        n = int(sys.argv[1])
    except IndexError:
        print("usage: %s [n]" % sys.argv[0])
        sys.exit(0)

    printroots(bitreverse(genroots(n)))

if __name__ == "__main__":
    main()
```

6.9 Main Loop

```
/*
 * mainloop.c
 *
 * Main loop for the dog bowl.
 *
 * This file contains the main loop for controlling access to the dog bowl. It
 * acts like a finite state machine, opening the bowl only when an object is
 * nearby and the FFT analysis gets a match. The bowl is then closed once the
 * proximity sensors are no longer active.
 *
 * Revision History:
 *      05 Jun 2015     Brian Kubisiak      Initial revision.
 */

#include <avr/interrupt.h>

#include "adc.h"
```

```
#include "data.h"
#include "fft.h"
#include "proximity.h"
#include "pwm.h"


/*
 * state
 *
 * Description: Data type for representing the current state of the dog bowl. It
 *              can be waiting for input, performing Fourier analysis, opening
 *              the bowl, or reseting the data.
 *
 * Notes:       This is used by the main loop FSM for determining what to do
 *              with hardware events. To see the transitions, read the
 *              documentation or the switch statement in the main loop.
 */
typedef enum _state {
    INIT_STATE, FFT_STATE, OPEN_STATE, RESET_STATE
} state;


/*
 * main
 *
 * Description: The main loop for the program is a finite state machine that
 *              polls the inputs (the data buffer and proximity sensors) to
 *              determine when to transition between states. For a full
 *              description of the FSM, see the documentation.
 *
 * Notes:       This main loop is not very efficient because it polls the inputs
 *              and runs some of the functions several times in a row.
 */
int main(void)
{
    state curstate = INIT_STATE;
    complex *buf;

    /* Initialize the peripherals used by the main loop. */
    init_adc();
    init_prox_gpio();
    init_pwm();

    DDRC = 0xFF;

    /* Turn on interrupts. */
    sei();

    /* Loop forever, until reset is applied or power is take away. */
    for (;;)
    {
        /* Determine the actions to perform as well as the next state based on
         * the current state. */
        switch (curstate)
        {
        case INIT_STATE:
```

```
            /* If data is ready and the proximity sensors are tripped, start the
             * data analysis. */
            if (is_data_collected() && is_obj_nearby()) {
                curstate = FFT_STATE;
            }
            /* If the data is ready, but the sensors are not tripped, then we
             * can ignore the noise. Reset the buffer and start waiting again.
             */
            else if (is_data_collected() && !is_obj_nearby()) {
                curstate = RESET_STATE;
            }
            /* Else, data is not collected; keep waiting in this state. */
            break;
        case FFT_STATE:
            /* Get the buffer of data and perform an FFT on the data. */
            buf = adc_get_buffer();
            fft(buf);

            /* Check that the recorded frequency spectrum matches the stored
             * spectrum. */
            if (is_fft_match(buf)) {
                /* If the spectrum matches, open the bowl. */
                curstate = OPEN_STATE;
            }
            else {
                /* Else, reset the data. */
                curstate = RESET_STATE;
            }
            break;
        case OPEN_STATE:
            /* Open the bowl after identifying the dog. */
            pwm_open();

            /* Wait until the proximity sensors are no longer tripped before
             * closing the bowl. */
            if (!is_obj_nearby()) {
                curstate = RESET_STATE;
            }
            break;
        case RESET_STATE:
            /* Close the dog bowl. */
            pwm_close();

            /* Reset the data collection. */
            adc_reset_buffer();

            /* Go back to waiting for data. */
            curstate = INIT_STATE;
            break;
        default:
            /* By default, go to the initial state. This should never happen. */
            curstate = INIT_STATE;
            break;
        }
    }
```

```
    return 0;
}
```

6.10 Proximity Sensor Function Declarations

```
/*
 * proximity.h
 *
 * Code for detecting proximity of the dog using the GPIO pins.
 *
 * This file contains an interface for detecting whether or not there is an
 * object detected by the ultrasonic rangefinders. The rangefinders feed into
 * comparators, which are connected to GPIO pins on the microcontroller; the
 * functions in this file will read the status off the GPIO pins to determine
 * whether or not anything is detected by the ultrasonic rangefinders.
 *
 * Peripherals Used:
 *      GPIO
 *
 * Pins Used:
 *      PA[0..7]
 *
 * Revision History:
 *      05 Jun 2015     Brian Kubisiak      Initial revision.
 *      08 Jun 2015     Brian Kubisiak      Changed polarity of signals.
 */

#ifndef _PROXIMITY_H_
#define _PROXIMITY_H_


/*
 * init_prox_gpio
 *
 * Description: This function initializes the GPIO pins so that they are ready
 *              to read data from the proximity sensors. This process involves:
 *               - Writing a 0 to DDA to set GPIO as input.
 *               - Writing a 1 to PORTA to enable the pull-up resistor.
 *               - Writing a 0 to the PUD bit in MCUCR
 *
 * Notes:       This function assumes that no other peripherals are going to use
 *              PA[0..7] pins; the configuration might not work if this is the
 *              case.
 */
void init_prox_gpio(void);


/*
 * is_obj_nearby
 *
 * Description: Determine whether or not there is an object nearby using the
 *              ultrasonic proximity sensors. This function returns 0 if no
 *              objects are detected by the proximity sensors, and returns
 *              nonzero is objects are detected.
```

```
 *
 * Return:      If no objects are in range of the ultrasonic sensors, returns 0.
 *              If one or more objects are in range, returns nonzero.
 *
 * Notes:       The return value uses the even 4 bits to represent the
 *              rangefinders in each direction, so the number of triggered
 *              rangefinders can also be found from the return value.
 */
unsigned char is_obj_nearby(void);


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

6.11 Proximity Sensor Functions

```
/*
 * proximity.c
 *
 * Code for detecting proximity of the dog using the GPIO pins.
 *
 * This file contains an interface for detecting whether or not there is an
 * object detected by the ultrasonic rangefinders. The rangefinders feed into
 * comparators, which are connected to GPIO pins on the microcontroller; the
 * functions in this file will read the status off the GPIO pins to determine
 * whether or not anything is detected by the ultrasonic rangefinders.
 *
 * Peripherals Used:
 *      GPIO
 *
 * Pins Used:
 *      PA[0..7]
 *
 * Revision History:
 *      05 Jun 2015     Brian Kubisiak      Initial revision.
 *      08 Jun 2015     Brian Kubisiak      Changed polarity of signals.
 */

#include <avr/io.h>

#include "proximity.h"

/* Constants for setting the values of the control registers. */
#define DDR_INPUT   0x00    /* Configure all pins as inputs. */
#define PORT_PULLUP 0xFF    /* Activate all pull-up resistors. */

/* Constant for masking out the unused pins. */
#define ACTIVE_PINS 0x55

/*
 * init_prox_gpio
 *
 * Description: This function initializes the GPIO pins so that they are ready
 *              to read data from the proximity sensors. This process involves:
 *               - Writing a 0 to DDA to set GPIO as input.
```

```
 *               - Writing a 1 to PORTA to enable the pull-up resistor.
 *
 * Notes:       This function assumes that no other peripherals are going to use
 *              PA[0..7] pins; the configuration might not work if this is the
 *              case.
 */
void init_prox_gpio(void)
{
    /* Set the configurations for IO port A. */
    DDRA  = DDR_INPUT;
    PORTA = PORT_PULLUP;
}


/*
 * is_obj_nearby
 *
 * Description: Determine whether or not there is an object nearby using the
 *              ultrasonic proximity sensors. This function returns 0 if no
 *              objects are detected by the proximity sensors, and returns
 *              nonzero is objects are detected.
 *
 * Return:      If no objects are in range of the ultrasonic sensors, returns 0.
 *              If one or more objects are in range, returns nonzero.
 *
 * Notes:       The return value uses the even 4 bits to represent the
 *              rangefinders in each direction, so the number of triggered
 *              rangefinders can also be found from the return value.
 */
unsigned char is_obj_nearby(void)
{
    /* Returns 0 if all pins are inactive, otherwise returns nonzero. */
    return ACTIVE_PINS & (~PINA);
}
```

6.12 Servo Control Function Declarations

```
/*
 * pwm.h
 *
 * Open and close the dog bowl with PWM.
 *
 * This file contains functions for using the PWM module for opening and closing
 * the dog bowl. The PWM port is connected to a servo motor, so changing the PWM
 * to a specific duty cycle will move the servo to the corresponding position.
 *
 * Peripherals Used:
 *      PWM
 *
 * Pins Used:
 *      PB7
 *
 * Revision History:
 *      05 Jun 2015     Brian Kubisiak      Initial revision.
```

```
 */

#ifndef _PWM_H_
#define _PWM_H_


/*
 * init_pwm
 *
 * Description: Initializes the PWM controlling the servo motor for
 *              opening/closing the dog bowl. This will set up the PWM pin for
 *              use and initialize the servo to a closed position.
 *
 * Notes:       The PWM uses the pin PB7; using this pin somewhere else could
 *              cause problems.
 */
void init_pwm(void);

/*
 * pwm_open
 *
 * Description: Open the dog bowl by moving the servo to a set open position.
 *              The servo is controlled by changing the duty cycle on the PWM
 *              output.
 *
 * Notes:       This function can be called multiple times in a row with no
 *              effect.
 */
void pwm_open(void);

/*
 * pwm_close
 *
 * Description: Close the dog bowl by moving the servo to a set closed position.
 *              The servo is controlled by changing the duty cycle on the PWM
 *              output.
 *
 * Notes:       This function can be called multiple times in a row with no
 *              effect.
 */
void pwm_close(void);


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

6.13 Servo Control with PWM

```
/*
 * pwm.c
 *
 * Open and close the dog bowl with PWM.
 *
 * This file contains functions for using the PWM module for opening and closing
 * the dog bowl. The PWM port is connected to a servo motor, so changing the PWM
```

```
 * to a specific duty cycle will move the servo to the corresponding position.
 *
 * Peripherals Used:
 *      PWM
 *
 * Pins Used:
 *      PB7
 *
 * Revision History:
 *      05 Jun 2015     Brian Kubisiak      Initial revision.
 */

#include <avr/io.h>

#include "pwm.h"

#define BOWL_OPEN   30
#define BOWL_CLOSED 15

/*
 * init_pwm
 *
 * Description: Initializes the PWM controlling the servo motor for
 *              opening/closing the dog bowl. This will set up the PWM pin for
 *              use and initialize the servo to a closed position.
 *
 * Notes:       The PWM uses the pin PB7; using this pin somewhere else could
 *              cause problems.
 */
void init_pwm(void)
{
    DDRB = 0x80;    /* Enable output on the PWM pin. */
    TCCR0A = 0x83;  /* Set pin to fast PWM mode. */
    TCCR0B = 0x05;  /* Prescale clock by 1024. */

    /* Start out with the bowl closed. */
    pwm_close();
}

/*
 * pwm_open
 *
 * Description: Open the dog bowl by moving the servo to a set open position.
 *              The servo is controlled by changing the duty cycle on the PWM
 *              output.
 *
 * Notes:       This function can be called multiple times in a row with no
 *              effect.
 */
void pwm_open(void)
{
    /* Set the new PWM compare value. */
    OCR0A = BOWL_OPEN;
}

/*
```

```
 * pwm_close
 *
 * Description: Close the dog bowl by moving the servo to a set closed position.
 *              The servo is controlled by changing the duty cycle on the PWM
 *              output.
 *
 * Notes:       This function can be called multiple times in a row with no
 *              effect.
 */
void pwm_close(void)
{
    /* Set the new PWM compare value. */
    OCR0A = BOWL_CLOSED;
}
```