Lab 2 Report

Deniz Alpay
1650403
deniza@uw.edu

Aidan Johnson
1431797
johnsj96@uw.edu

17 April 2018

EE 443, Spring 2018
Design and Application of Digital Signal Processors
Department of Electrical Engineering
University of Washington

Problem 1: Echo Effect

The first lab problem entailed generating a superimposed echo effect for an input audio signal. The DIP switches SW5, SW6, and SW7 were used to control the echo effect. SW5 enabled the echo effect and the four combinations of SW6 and SW7 controlled the number of buffer (delay) samples. When SW5 was disabled, just an unaffected input was written as the output. Implementing this effect relied on storing a user set number of samples of the input in a buffer, and then adding the buffer contents to the output signal. This, as one would expect, resulted in an echoed input signal. The time difference between the current input/output (i.e., the original input signal) and echo output is a function of the buffer length.

Per specification, a 16 kHz input sampling frequency and five buffer sample lengths (0, 200, 400, 600, and 800 samples) were set. However, to enhance the echoing, we took the liberty of increasing the sample lengths by a factor of 10. The specified samples buffer lengths would result in a at most 0.05 s time delay between the signal and its echo. We were unable to perceive this time delay when music was an input (instead of a MIC input). Increasing the buffer length increased the time delays from 0.0125, 0.025, 0.0375 and 0.05 s to 0.125, 0.25, 0.375, and 0.5 s delays, which were much more observable. To aide the user, we also chose to indicate the switch position with three LEDs (D4, D5, and D6). As such, our `ISRs.c` C code below reflects our design modification.

```c
#include "DSP_Config.h"

#define LEFT  0
#define RIGHT 1
#define MAX_DELAY 8000

volatile union {
    Uint32 UINT;
    Int16 Channel[2];
} CodecDataIn, CodecDataOut;

float xLeft, xRight, yLeft, yRight;
int idx = 0; // index for buffer value
float buffer[2][MAX_DELAY + 1]; // space for left + right
int delaySamples = 0; // number of samples to delay by


float getDelayedVals(int k)
{
    float delayedVal;

    // For left channel, k = 0 and for right channel, k = 1
    int ind; // Index of the delayed value in the buffer
    if (delaySamples > idx) {
        ind = MAX_DELAY - (delaySamples - idx);
    } else {
        ind = idx - delaySamples;
    }
    delayedVal = buffer[k][ind];

    return delayedVal;
}

interrupt void Codec_ISR()
{
    if(CheckForOverrun()) // overrun error occurred (halted DSP)
        return; // so serial port is reset to recover

    CodecDataIn.UINT = ReadCodecData(); // get input data samples

    Int32 SW = ReadSwitches();

    // check switches: SW6 = 2^1, SW7 = 2^2; 0 = up, 1 = down
    // LEDs: D4 = 2^0, D5 = 2^1, D6 = 2^2; 0 = off, 1 = on
    if(SW == 2) { // SW5 SW7 up, SW6 down: D4 D6 on, D5 off
        delaySamples = 600;
        WriteLEDs(5);
    }
    else if(SW == 4) { // SW5 SW6 up, SW7 down: D4 D5 on, D6 off
        delaySamples = 400;
        WriteLEDs(3);
    }
    else if(SW == 6) { // SW5 up, SW6 SW7 down: D4 D6 on, D5 off
        delaySamples = 800;
        WriteLEDs(1);
    }
    else if(SW == 0) { // SW5 SW6 SW7 up: D4 D5 D6 on
        delaySamples = 200;
        WriteLEDs(7);
    }
    else  {    // SW5 down: D4 D5 D6 off
        delaySamples = 0;
        WriteLEDs(0);
    }

    xLeft = CodecDataIn.Channel[LEFT]; // current LEFT input value
    xRight = CodecDataIn.Channel[RIGHT]; // current RIGHT input value

    buffer[LEFT][idx] = xLeft;
    buffer[RIGHT][idx] = xRight;
    if (++idx >= MAX_DELAY) // implement circular buffer
        idx = 0;

    float leftDelay = getDelayedVals(LEFT);
    float rightDelay = getDelayedVals(RIGHT);
    if (delaySamples == 0) {
        yLeft = xLeft;
        yRight = xRight;
    }
    else {
        yLeft = xLeft + leftDelay;
        yRight = xRight + rightDelay;
    }

    CodecDataOut.Channel[LEFT] = yLeft; // setup the LEFT value
    CodecDataOut.Channel[RIGHT] = yRight; // setup the RIGHT value

    WriteCodecData(CodecDataOut.UINT); // send output data to port
}
```

Problem 2: Assembly 1

As the first TI C6x assembly language problem assigned thus far, the specification was intended to force us to familiarize ourselves with this low-level programming language. In this problem, we were to implement assembly code to flip an input data set recorded from the MIC (or even the audio in jack) and stored as an array of data with a fixed length (defined by a global parameter). In a high-level programming and object-oriented language (e.g. Java) this operation could easily be performed with a stack (first in, last out or FILO) data structure, where the input data is pushed to the first stack and then popped to an output array. Taking inspiration from the stack, we set out to implement a stack in assembly.

Our `stack.asm` file is as follows. Comments explaining the algorithm follow the ";". The starting register addresses and values are as follows: A4 stores the address in memory that points to the the front of the input array, B4 stores the number of values in the input array, A6 stores the size of the data values in terms of Bytes, and B6 stores the address in memory that points to the front of the output array.

```
.def _stack

_stack:
               MV    B4,B1              ;push loop count
               MV    B4,A1              ;pop loop count
               MV    A4,B15             ;stack pointer -> front of input array

push:                                   ;push input into stack with stack pointer
              LDW    *B15++,A6          ;assumes 4 Byte float i/o given by A6
              SUB    B1,1,B1            ;decrement count
       [B1]   B      push               ;branch to loop if count != 0
              NOP    5

pop:                                    ;pops stack to output array
```

```
        LDW     *--B15,A0           ;popped by stack pointer
        NOP     5
        STW     A0,*B6++            ;popped to output array pointer
        SUB     A1,1,A1             ;decrement count
[A1]    B       pop
        NOP     5
        B       B3                  ;returns
        NOP     5
```

In the C language `ISRs.c` file, `stack.asm` is called to perform the flipping operation. It is as follows:

```c
#include "DSP_Config.h"

#define LEFT  0
#define RIGHT 1
#define NUM_SAMPLES 512

volatile float input[NUM_SAMPLES];
volatile float output[NUM_SAMPLES];
int bytes = sizeof(float);
int itr = 0;

volatile union {
    Uint32 UINT;
    Int16 Channel[2];
} CodecDataIn, CodecDataOut;

interrupt void Codec_ISR()
{
    if(CheckForOverrun()) // overrun error occurred (halted DSP)
        return; // so serial port is reset to recover

    CodecDataIn.UINT = ReadCodecData(); // get input data samples

    if (itr < NUM_SAMPLES) {
        input[itr] = CodecDataIn.Channel[LEFT];
        output[itr] = 0;
    } else if (itr == NUM_SAMPLES) {
        stack(input, NUM_SAMPLES, bytes, output);
        printArrays(input, output); // prints arrays to console
    }
    itr++;

    WriteCodecData(0); // send output data to port
}
```

For example if instead the input array consists of 10 indices of the iterator, this would be printed to console as:

```
x[n] = [ 0.00 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 10.00 ]
x^[n] = [ 10.00 9.00 8.00 7.00 6.00 5.00 4.00 3.00 2.00 1.00 0.00 ]
```

Memory constraints withstanding, an input array consisting of data recorded from the MIC or any other input, regardless of array size, will perform equivalently.

Problem 3: Assembly 2

In problem 3, the autocorrelation function was implemented in both the C language and the C6x assembly language. By implementing the autocorrelation in a low-level and high-level language, we hoped to examine the processing time differences between the two implementations. While the problem specified that the first 20 coefficients of a 1024 sample signal be computed, we found that memory limits were interfering with the computation. As a result, we decreased the number of autocorrelation coefficients to 5 and the signal length to 10.

The C6x assembly implementation of the autocorrelation function, named xcorr.asm is as follows. The register A4 stores the address in memory that points to the the front of the input signal array, B4 stores the number of samples in the input signal array, A6 stores the number of autocorrelation coefficients in the output array, and B6 stores the address in memory that points to the front of the output array of autocorrelation coefficients.

```
               .def _xcorr

_xcorr:                                  ;auto- (cross- with self) correlation
                MV B4,A0                 ;N samples counter, n
                MV A6,B0                 ;K coefficient or shift counter, k
                MV A4,A3                 ;resets n+k to n
                MV A6,A5                 ;backup coefficient counter

corr:
                MV A4,A5                 ;x(n) pointer reset
                MV A4,A3                 ;x(n+k) pointer reset
                MV B4,A0                 ;resets samples counter
                SUB A0,B0,A0             ;shifts N samples by k

shift:                                   ;sets x(n+k) pointer
                ADD A3,4,A3              ;shifts by next k
                SUB B0,1,B0              ;decrements shift
    [B0]    B     shift
                NOP 5

                ZERO A1                   ;temp storage for each product
                ZERO B1                   ;temp storage for sum of products
                ZERO A2                 ;temp storage for multiplier
                ZERO B2                 ;temp storage for multiplicand

sum:                                    ;correlation summation
                LDW *A5++,A2           ;multiplier
                NOP 8
                LDW *A3++,B2           ;multiplicand
                NOP 8
                MPYSP A2,B2,A1         ;multiplies in1 and in2 at n and n+k sample index
                NOP 8
                ADDSP B1,A1,B1         ;sums product
                NOP 8
                SUB A0,1,A0            ;decrements samples counter
                NOP 8
    [A0]        B sum
                NOP 5

                STW B1,*B6++           ;stores coefficient, increments to next store
                NOP 8
                SUB A6,1,A6            ;decrements coefficient counter
                MV A6,B0               ;resets shift counter
                NOP 8
    [B0]        B    corr
                NOP 5

                B     B3                ;returns
                NOP   5
```

In the C language `ISRs.c` file, `xcorr.asm` is called to perform the flipping operation. The code for this call is as follows:

```
#include "DSP_Config.h"
#include <time.h>

#define LEFT  0
#define RIGHT 1

#define NUM_SAMPLES 10
#define NUM_AUTOCORR 5

volatile float input[NUM_SAMPLES];
volatile float output[NUM_AUTOCORR];
volatile float coeff[NUM_AUTOCORR];
int itr = 0;

volatile union {
    Uint32 UINT;
    Int16 Channel[2];
```

```c
} CodecDataIn, CodecDataOut;

void autocorrelation()
{
    int k, n;
    for (k = 1; k <= NUM_AUTOCORR; k++) {
        for (n = 0; n < NUM_SAMPLES - k; n++) {
                output[NUM_AUTOCORR - k] += input[n]*input[n + k];
        }
    }
}

interrupt void Codec_ISR()
{

        if(CheckForOverrun())                   // overrun error occurred (i.e. halted DSP)
        return;                                 // so serial port is reset to recover

      CodecDataIn.UINT = ReadCodecData(); // get input data samples

      time_t start, stop;

      if (itr < NUM_SAMPLES) {
        input[itr] = itr;                     //CodecDataIn.Channel[LEFT];
    } else if (itr == NUM_SAMPLES) {
        start = time(0);
        autocorrelation();
        //xcorr(input, NUM_SAMPLES, NUM_AUTOCORR, output);
        stop = time(0);
        time_t dur = stop - start;
        printArrays(input, output, dur);
      }
      itr++;


    WriteCodecData(0);
}
```

The function `printArrays` is defined in `main.c` as seen below.

```c
#include "DSP_Config.h"
#include <time.h>
#include <stdio.h>

#define NUM_SAMPLES 10                // Change to 1024
#define NUM_AUTOCORR 5                 // Change to 20

int main()
{
        DSP_Init();

        StartUp();

        while(1) {
          ;
        }
}

void printArrays(volatile float * input, volatile float * output, time_t dur)
{
    int i;
    printf("\nx[n] = [ ");
    for (i = 0; i < NUM_SAMPLES; i++) {
        printf("%.2f ", input[i]);
    }
    printf("]\n");

    printf("R[k] = [ ");
    for (i = 0; i < NUM_AUTOCORR; i++) {
        printf("%.2f ", output[i]);
    }
    printf("]\n");

    printf("Processing time: %f us\n", dur);

}
```

When running the above code using the assembly implementation, the following is printed to the console.

```
x[n] = [ 0.00 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 ]
R[k] = [ 80.00 115.00 154.00 196.00 240.00 ]
Processing time: 240.0000 us
```

While the C language implementation prints the following to the console.

```
x[n] = [ 0.00 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 ]
R[k] = [ 80.00 115.00 154.00 196.00 240.00 ]
Processing time: 240.0000 us
```

Notice that the processing times are in microseconds and are also the same for both implementations. This indicates that, perhaps at least in this specific operation, there is little difference in time efficiency. A larger input array could give different results. To conclude, the outputs are identical and match the true autocorrelation. The specific MATLAB code (and for free and open source Scilab, in which the lines of code are identical) for confirming this autocorrelation calculation is: `x = [0:9]; y = xcorr(x, x, 5); y = y(1:5).`

Problem 4: Delay Based Pitch Shifter

While the a pitch shift is a change in the frequency domain, in this problem a time domain pitch shifter was implemented in the C language. Both an upward and downward pitch shift was implemented, which was controlled by switch SW5. The pitch shifter could be toggled by switch SW6 to refer to the original audio. The pitch shift involves a continually changing delays and gains on two channels made from a single channel input. A buffer of 700 samples was used as this was the maximum delay for either channel. The effect of the pitch shifter is very noticable for speech, but less obvious when listening to very slow or fast music.

As a demonstration of the functionality of the pitch shifter, Figures 1, 2, and 3 show each of the toggle states for a 1 kHz sine wave input sent in through the 3.5 mm input jack. The oscilloscope waveforms clearly show a functioning pitch (frequency) shifter. Channel 2 of the oscilloscope is the input/output sine wave while Channel 1 is the pitch-shifted output.

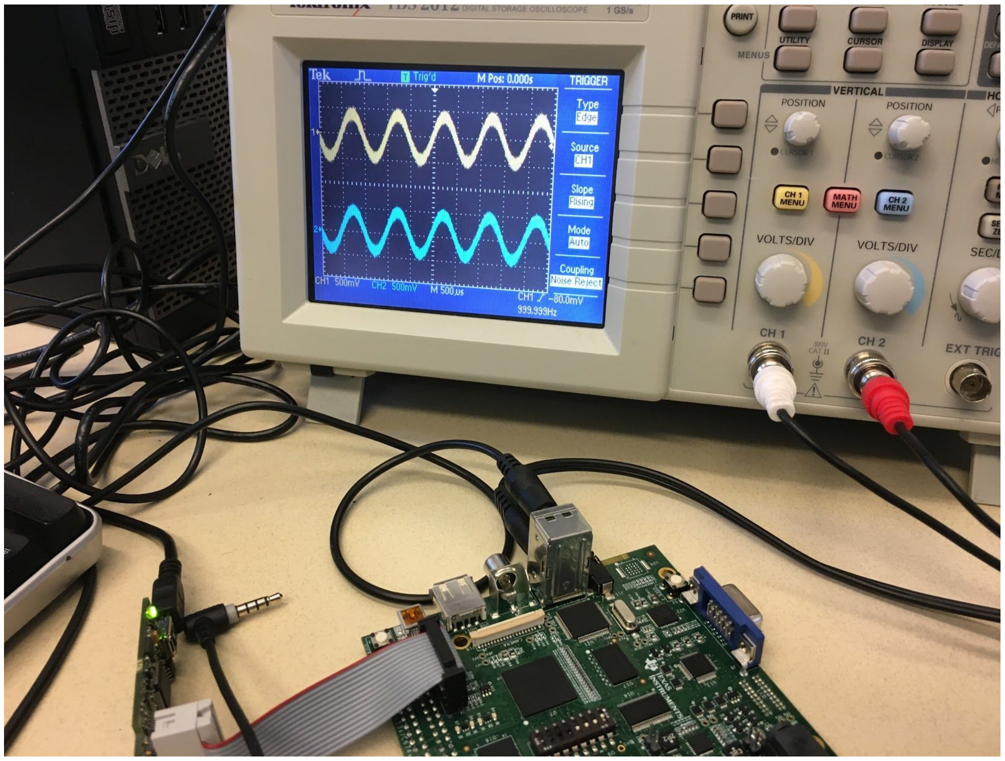

Figure 1: Oscilloscope output for 1 kHz sine wave input when pitch shift disabled (SW5 down).

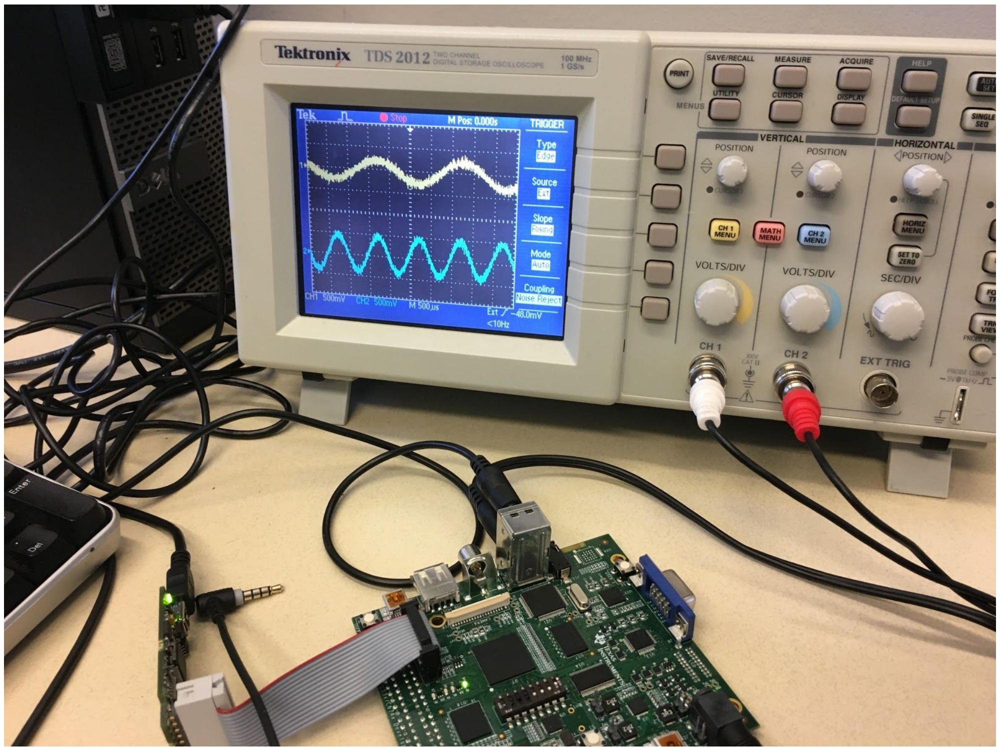

Figure 2: Oscilloscope output for 1 kHz sine wave input when pitch down enabled (SW6 down; LED D4 is on as an indicator for the shift enable state).

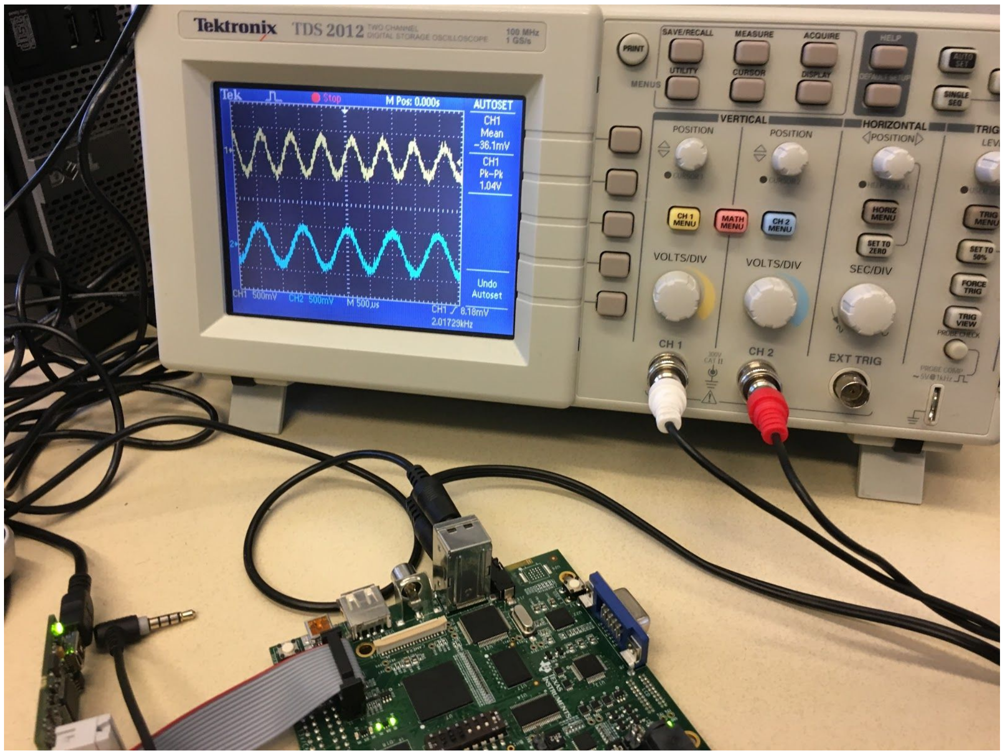

Figure 3: Oscilloscope output for 1 kHz sine wave input when pitch up enabled (with SW6 up; LED D5 is on as an indicator for the shift up state).

The C language code below controls the pitch shifter, allowing us to toggle the pitch shift as described above.

```c
#define LEFT  0
#define RIGHT 1
#define A 0                                             // Index for channel A
#define B 1                                             // Index for channel B
#define MAX_DELAY 700
#define BIAS 650

int idx = 0;                                      // Index of the end of the buffer
float channDelay[2];                             // Delay of each channel
float channGain[2];                               // Gain of each channel
int channGainInc[2] = {1, 0};   // 1 if channel gain is increasing, 0 if decreasing
int upwardPitch = 1;             // 1 if upward pitch change, 0 if downward pitch change
int pitchShiftIsOn = 1;         // 1 if pitch shifter is enabled, 0 is disabled
int prevSW = -1;                                  // Previous switch values
float buffer[MAX_DELAY + 1];                      // 0 to 700


volatile union {
    Uint32 UINT;
    Int16 Channel[2];
} CodecDataIn, CodecDataOut;

void updateDelays()
{
    // Channel A
    if (upwardPitch) {
        if (channDelay[A] <= 0) {
                channDelay[A] = (float)MAX_DELAY;
        } else {
                channDelay[A] -= 0.5;
        }
    } else {
        if (channDelay[A] >= (float)MAX_DELAY) {
                channDelay[A] = 0.0;
        } else {
                channDelay[A] += 0.5;
        }
    }

    // Channel B
    if (channDelay[A] >= (float)MAX_DELAY/2) {
        channDelay[B] = channDelay[A] - (float)MAX_DELAY/2;
    } else {
        channDelay[B] = channDelay[A] + (float)MAX_DELAY/2;
    }
}
```

```
void updateGains()
{
    // Channel A
    if (channGainInc[A]) {
        if (channGain[A] >= 1) {
                channGainInc[A] = 0;
                channGain[A] -= (float)1/MAX_DELAY;
        } else {
                channGain[A] += (float)1/MAX_DELAY;
        }
    } else {
        if (channGain[A] <= 0) {
                channGainInc[A] = 1;
                channGain[A] += (float)1/MAX_DELAY;
        } else {
                channGain[A] -= (float)1/MAX_DELAY;
        }
    }

    // Channel B
    channGain[B] = 1 - channGain[A];
}

float getDelayedVals(int k)
{
    float delayedVal;

    int ind;     // Index of the delayed value in the buffer
    if ((int) channDelay[k] > idx) {
        ind = MAX_DELAY - ((int) channDelay[k] - idx);
    } else {
        ind = idx - (int) channDelay[k];
    }
    delayedVal = buffer[ind];

    return delayedVal;
}

interrupt void Codec_ISR()
{
        if(CheckForOverrun())
        return;

      CodecDataIn.UINT = ReadCodecData();

      // Toggle the pitch shifter and upward/downward pitch change
      Int32 SW = ReadSwitches();

      if (SW != prevSW) {
        if (SW == 0) {                // SW5 up, SW6 up:
               upwardPitch = 1;
               pitchShiftIsOn = 1;
               channDelay[A] = (float)MAX_DELAY;
               channDelay[B] = (float)MAX_DELAY/2;
               channGain[A] = 0.0;
               channGain[B] = 1.0;
               WriteLEDs(3);
        } else if (SW == 1) {      // SW5 down, SW6 up:
               upwardPitch = 1;
               pitchShiftIsOn = 0;
               WriteLEDs(2);
        } else if (SW == 2) {     // SW5 up, SW6 down:
               upwardPitch = 0;
               pitchShiftIsOn = 1;
               channDelay[A] = 0.0;
               channDelay[B] = (float)MAX_DELAY/2;
               channGain[A] = 0.0;
               channGain[B] = 1.0;
               WriteLEDs(1);
        } else if (SW == 3) {      // SW5 down, SW6 down:
               upwardPitch = 0;
               pitchShiftIsOn = 0;
               WriteLEDs(0);
        }

        prevSW = SW;
      }

      // Update delays and gains
      updateDelays();
      updateGains();

      // Update buffer
      buffer[idx++] = CodecDataIn.Channel[LEFT];
      if (idx > MAX_DELAY)
        idx = 0;

    if (pitchShiftIsOn) {
        float channADelayed = getDelayedVals(A);
        float channBDelayed = getDelayedVals(B);
        CodecDataOut.Channel[LEFT] = channGain[A]*channADelayed + channGain[B]*channBDelayed
- 2*BIAS;
    } else {
        CodecDataOut.Channel[LEFT] = CodecDataIn.Channel[LEFT];
    }
    CodecDataOut.Channel[RIGHT] = CodecDataIn.Channel[RIGHT];

    WriteCodecData(CodecDataOut.UINT);
}
```