# Lab 7

## FFT, Fast Convolution, and Spectral Analysis

### ECE 406: Real-Time Digital Signal Processing

April 27, 2015

Christopher Daffron

cdaffron@utk.edu

There were several objectives for this lab. The first objective was to get familiar with performing convolution in the frequency domain using FFT. A secondary objective was to also use the windowing technique for the purpose of spectral analysis.

# Task 0

The first task given for this lab assignment was to manually trace through figure 2 in the lab instructions and turn that trace in with the rest of the assignment. That tracing is shown below.

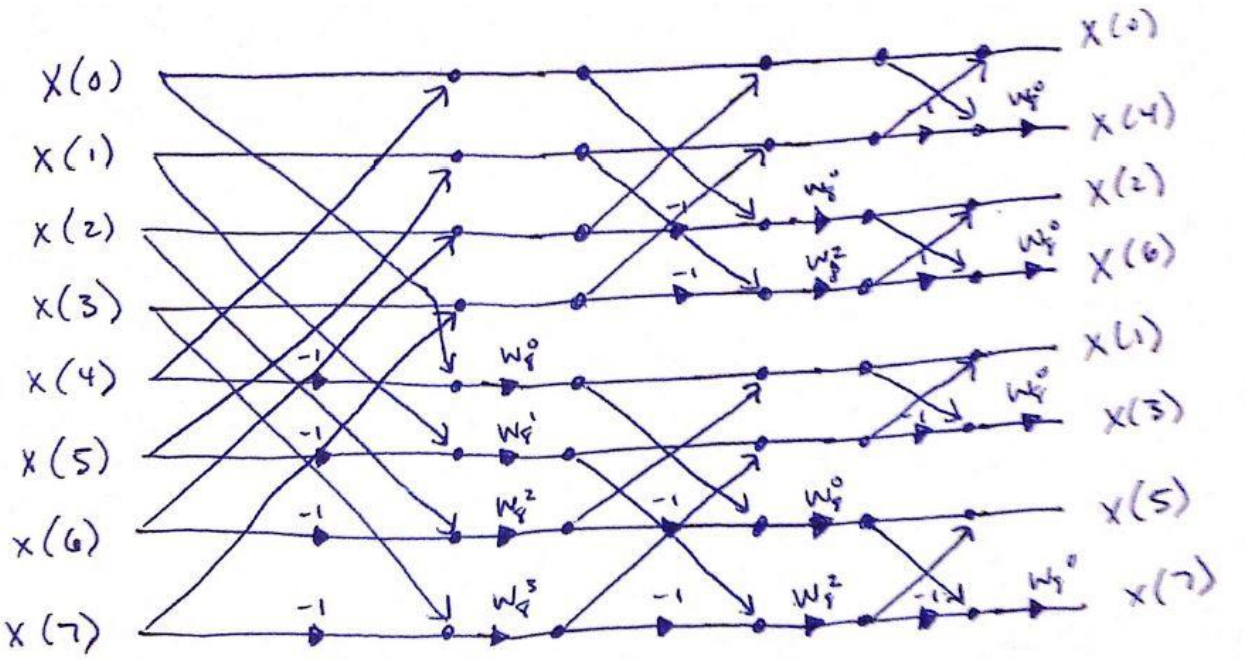


Figure 1: FFT Butterfly Diagram

# Task 1

## Step 1

Step 1 of this task was to create a frame-based implementation of the FIR frame-based filter used in Lab 6. The first task of doing this was to convert the non-DSK implementation of the FFT provided by the book to one that is compatible with the DSK board. This was relatively trivial as the code was able to be inserted directly into a DSK project. Next, the size of the FFT needed to be computed, while keeping in mind that the length of the FFT must be a power of 2 and must be equal to the length of the filter plus the length of the input frame minus one. My filter has a length of 60 and I chose a FFT length of 1024, so my input frame needs to contain 965 samples. After determining these lengths, I implemented the FFT operations necessary to filter the input, utilizing the overlap-add technique to ensure that linear convolution is done instead of circular convolution. To improve the performance of the design, I only zero-pad the filter coefficients and compute the FFT once during initialization since this data will remain constant for all frames. I then implemented the complex multiplication, the IFFT, and the overlap-add operations to complete the frequency domain filtering. Upon testing this design, I found that it behaves as expected.

## Step 2

Step 2 of this task was to sweep the frequency inputs and create a chart comparing the expected and actual frequency response for the filter. As it has in previous labs, this filter behaved as expected and produced the results shown below.

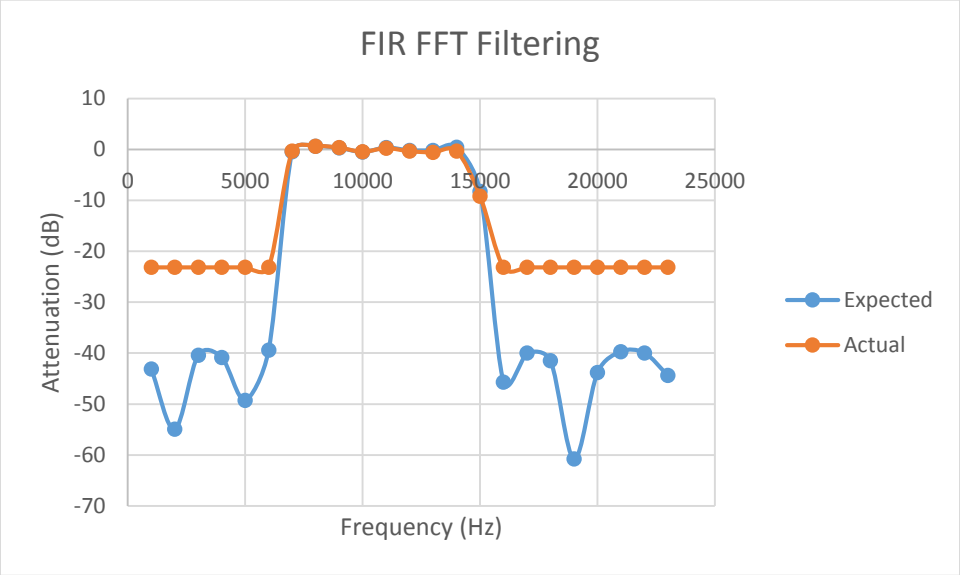


Figure 2: FFT Filtering Frequency Response

## Step 3

The next task was to compare the number of clock cycles required for the fast, frequency domain convolution versus the frame-based time domain convolution from the previous lab. For the time domain convolution, 3,500 cycles were required per sample with optimization disabled and 262 cycles were required per sample with optimization enabled. For the frequency-domain fast convolution, 2,266 cycles were required per sample with optimization disabled and 458 cycles were required per sample with optimization enabled. This result is interesting as it shows that when comparing the raw algorithms as they are implemented, the frequency domain convolution is more efficient, but the compiler is better at optimizing the time domain convolution.

## Step 4

The final step of Task 1 was to compare the spectra obtained using the FFT vs using the FFT with windowing applied. After consulting with Mo, it was decided to do this step solely in MATLAB. For this step, I compared the results of the raw FFT with an FFT with a Hanning window applied. To get a somewhat spread spectrum input, I used an audio file of a gong as my input.

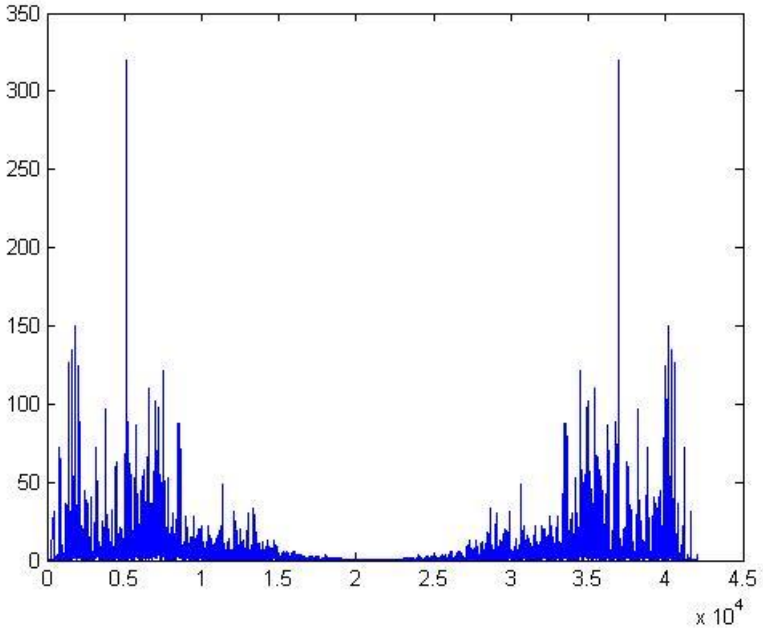


Figure 3: FFT Response (Hanning Window)

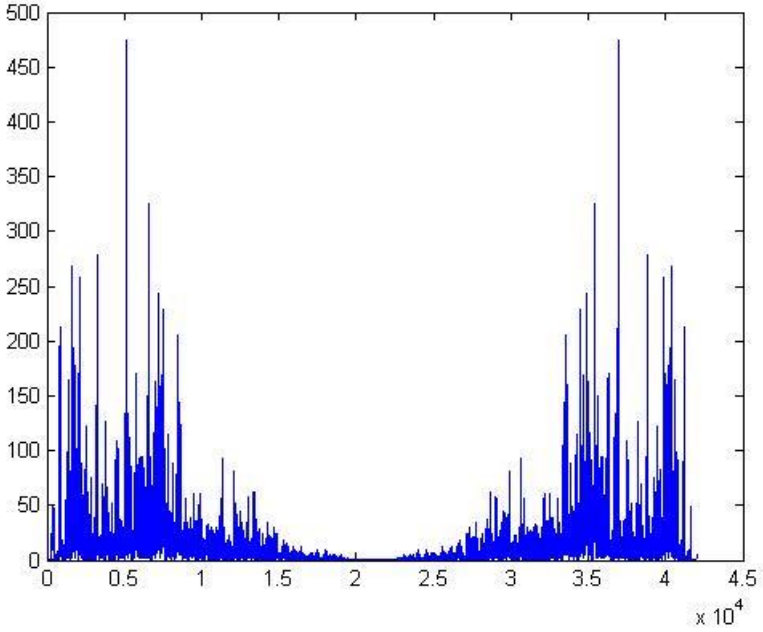


Figure 4: FFT Response (No Window)

As is shown in the figures, the frequency response is nearly identical with the exception of the magnitude of the frequency response. By changing the magnitudes, the lower peaks aren't overshadowed as much by the highest peak. This is the same reason that FFT responses are usually viewed using a logarithmic scale.

# Lab Report Questions

Many more operations are required for the DFT than the FFT. The DFT requires $n^2$ operations, so it requires 16,777,216 operations for a 4096-point DFT. By comparison, the FFT requires n log(n) operations, so it requires 49,152 operations for the 4096-point FFT. This shows why the FFT is so much more efficient than the DFT as the DFT requires 341 times as many operations. Windowing needs to be applied when performing spectral analysis so that all of the possible frequencies are represented equally in the scaling of the final output. This is the same reason that non-windowed spectrums are generally displayed using a logarithmic scale.

# Appendix

## Coeff.c

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011

/* coeff.c                              */
/* FIR filter coefficients              */
/* exported by MATLAB using FIR_DUMP2C */


/* Equiripple FIR LPF with passband to */
/* 5 kHz assuming Fs=48 kHz            */


#include "coeff.h"

float B[N_coeff] = {
-0.0049123250992086183,
-0.014619079916495251,
 0.0060757570503416243,
 0.01952535804916462,
 0.0024331187695853683,
-0.0098746099102310358,
-0.0037963320758576859,
-0.012010346542919192,
-0.01035742425347112,
 0.01992326384135619,
 0.021487608632075595,
-0.0036831076567773926,
-0.0025285456210165994,
-0.010532837642683505,
-0.034386903509944826,
-0.0011850529678652522,
 0.039757652237396482,
 0.014026994706631228,
 0.0043585313963652086,
 0.014292489095961029,
-0.04562980713703095,
-0.062041754213086692,
 0.02997771785352504,
 0.041487703139491801,
```

```
 0.0066936154158850807,
 0.087583480119140356,
 0.030241608546776121,
-0.23544107929841201,
-0.15849062460484048,
 0.26627195921759944,
 0.26627195921759944,
-0.15849062460484048,
-0.23544107929841201,
 0.030241608546776121,
 0.087583480119140356,
 0.0066936154158850807,
 0.041487703139491801,
 0.02997771785352504,
-0.062041754213086692,
-0.04562980713703095,
 0.014292489095961029,
 0.0043585313963652086,
 0.014026994706631228,
 0.039757652237396482,
-0.0011850529678652522,
-0.034386903509944826,
-0.010532837642683505,
-0.0025285456210165994,
-0.0036831076567773926,
 0.021487608632075595,
 0.01992326384135619,
-0.01035742425347112,
-0.012010346542919192,
-0.0037963320758576859,
-0.0098746099102310358,
 0.0024331187695853683,
 0.01952535804916462,
 0.0060757570503416243,
-0.014619079916495251,
-0.0049123250992086183,
};
```

# fft.c

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011

// This code calculates the fft of an N point complex data sequence, x[N].

//   The fft of real input values can be calculated by omitting the
//   x[].imag declarations. The fft results (the N complex numbers) are
//   returned in x[N]. This algorithm is based on the discussion in,

//   "C Algorithms for Real-Time DSP", by Paul M. Embree
//   Prentice-Hall PTR, copyright 1995.

#include <math.h>
#include <stdio.h>
#include "fft.h"
```

```
#define PI 3.14159265358979323846

void fft_c(int n, COMPLEX *x, COMPLEX *W)
{
    COMPLEX u, temp, tm;
    COMPLEX *Wptr;

    int i, j, k, len, Windex;

       /* start fft */
    Windex = 1;
    for(len = n/2 ; len > 0 ; len /= 2) {
        Wptr = W;
        for (j = 0 ; j < len ; j++) {
            u = *Wptr;
            for (i = j ; i < n ; i = i + 2*len) {
                temp.real = x[i].real + x[i+len].real;
                temp.imag = x[i].imag + x[i+len].imag;
                tm.real = x[i].real - x[i+len].real;
                tm.imag = x[i].imag - x[i+len].imag;
                x[i+len].real = tm.real*u.real - tm.imag*u.imag;
                x[i+len].imag = tm.real*u.imag + tm.imag*u.real;
                x[i] = temp;

            }
            Wptr = Wptr + Windex;
        }
        Windex = 2*Windex;
    }

       /* rearrange data by bit reversing */
       j = 0;
       for (i = 1; i < (n-1); i++) {
              k = n/2;
              while(k <= j) {
                     j -= k;
                     k /= 2;
              }
              j += k;
              if (i < j) {
                     temp = x[j];
                     x[j] = x[i];
                     x[i] = temp;
              }
       }
}

void init_W(int n, COMPLEX *W)
{
    int i;

       float a = 2.0*PI/n;

    for(i = 0 ; i < n ; i++) {
        W[i].real = (float) cos(-i*a);
        W[i].imag = (float) sin(-i*a);
    }
}
```

```
void init_W_neg(int n, COMPLEX *W)
{
	int i;

	float a = 2.0*PI/n;

	for(i = 0; i < n; i ++) {
		W[i].real = (float) cos(-i*a);
		W[i].imag = (float) sin(-i*a);
	}
}
```

# ISRs.c

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011

////////////////////////////////////////////////////////////////////////
// Filename: ISRs.c
//
// Synopsis: Interrupt service routines for EDMA
//
////////////////////////////////////////////////////////////////////////

#include "DSP_Config.h"
#include "math.h"
#include "frames.h"
#include "coeff.h"

// frame buffer declarations
//#define BUFFER_COUNT            1024   // buffer length in McBSP samples (L+R)
#define BUFFER_COUNT       965
#define BUFFER_LENGTH      BUFFER_COUNT*2 // two shorts read from McBSP each time
#define NUM_BUFFERS        3     // don't change this!

#pragma DATA_SECTION (buffer, "CE0"); // allocate buffers in SDRAM
Int16 buffer[NUM_BUFFERS][BUFFER_LENGTH];
// there are 3 buffers in use at all times, one being filled from the McBSP,
// one being operated on, and one being emptied to the McBSP
// ready_index --> buffer ready for processing
volatile Int16 buffer_ready = 0, over_run = 0, ready_index = 0;

// fft defines
#include "fft.h"
//#define N BUFFER_COUNT
#define N 1024
COMPLEX x[N], W[N], NW[N];
COMPLEX coeffs[N];
COMPLEX mult_result[N];
float saved[N - BUFFER_COUNT];
float y[N];
float x_raw[N];
float B_pad[N];

void EDMA_Init()
////////////////////////////////////////////////////////////////////////
// Purpose:   Configure EDMA controller to perform all McBSP servicing.
```

```
//            EDMA is setup so buffer[2] is outbound to McBSP, buffer[0] is
//            available for processing, and buffer[1] is being loaded.
//            Conditional statement ensure that the correct EDMA events are
//            used based on the McBSP that is being used.
//            Both the EDMA transmit and receive events are set to automatically
//            reload upon completion, cycling through the 3 buffers.
//            The EDMA completion interrupt occurs when a buffer has been filled
//            by the EDMA from the McBSP.
//            The EDMA interrupt service routine updates the ready buffer index,
//            and sets the buffer ready flag which is being polled by the main
//            program loop
//
// Input:     None
//
// Returns:   Nothing
//
// Calls:     Nothing
//
// Notes:     None
////////////////////////////////////////////////////////////////////////
{
       EDMA_params* param;

       // McBSP tx event params
       param = (EDMA_params*)(EVENTE_PARAMS);
       param->options = 0x211E0002;
       param->source = (unsigned int)(&buffer[2][0]);
       param->count = (0 << 16) + (BUFFER_COUNT);
       param->dest = 0x34000000;
       param->reload_link = (BUFFER_COUNT << 16) + (EVENTN_PARAMS & 0xFFFF);

       // set up first tx link param
       param = (EDMA_params*)EVENTN_PARAMS;
       param->options = 0x211E0002;
       param->source = (unsigned int)(&buffer[0][0]);
       param->count = (0 << 16) + (BUFFER_COUNT);
       param->dest = 0x34000000;
       param->reload_link = (BUFFER_COUNT << 16) + (EVENTO_PARAMS & 0xFFFF);

       // set up second tx link param
       param = (EDMA_params*)EVENTO_PARAMS;
       param->options = 0x211E0002;
       param->source = (unsigned int)(&buffer[1][0]);
       param->count = (0 << 16) + (BUFFER_COUNT);
       param->dest = 0x34000000;
       param->reload_link = (BUFFER_COUNT << 16) + (EVENTP_PARAMS & 0xFFFF);

       // set up third tx link param
       param = (EDMA_params*)EVENTP_PARAMS;
       param->options = 0x211E0002;
       param->source = (unsigned int)(&buffer[2][0]);
       param->count = (0 << 16) + (BUFFER_COUNT);
       param->dest = 0x34000000;
       param->reload_link = (BUFFER_COUNT << 16) + (EVENTN_PARAMS & 0xFFFF);


       // McBSP rx event params
       param = (EDMA_params*)(EVENTF_PARAMS);
```

```
        param->options = 0x203F0002;
        param->source = 0x34000000;
        param->count = (0 << 16) + (BUFFER_COUNT);
        param->dest = (unsigned int)(&buffer[1][0]);
        param->reload_link = (BUFFER_COUNT << 16) + (EVENTQ_PARAMS & 0xFFFF);

        // set up first rx link param
        param = (EDMA_params*)EVENTQ_PARAMS;
        param->options = 0x203F0002;
        param->source = 0x34000000;
        param->count = (0 << 16) + (BUFFER_COUNT);
        param->dest = (unsigned int)(&buffer[2][0]);
        param->reload_link = (BUFFER_COUNT << 16) + (EVENTR_PARAMS & 0xFFFF);

        // set up second rx link param
        param = (EDMA_params*)EVENTR_PARAMS;
        param->options = 0x203F0002;
        param->source = 0x34000000;
        param->count = (0 << 16) + (BUFFER_COUNT);
        param->dest = (unsigned int)(&buffer[0][0]);
        param->reload_link = (BUFFER_COUNT << 16) + (EVENTS_PARAMS & 0xFFFF);

        // set up third rx link param
        param = (EDMA_params*)EVENTS_PARAMS;
        param->options = 0x203F0002;
        param->source = 0x34000000;
        param->count = (0 << 16) + (BUFFER_COUNT);
        param->dest = (unsigned int)(&buffer[1][0]);
        param->reload_link = (BUFFER_COUNT << 16) + (EVENTQ_PARAMS & 0xFFFF);

        *(unsigned volatile int *)ECR = 0xf000;   // clear all McBSP events
        *(unsigned volatile int *)EER = 0xC000;
        *(unsigned volatile int *)CIER = 0x8000; // interrupt on rx reload only
}

void ZeroBuffers()
////////////////////////////////////////////////////////////////////////
// Purpose:   Sets all buffer locations to 0
//
// Input:     None
//
// Returns:   Nothing
//
// Calls:     Nothing
//
// Notes:     None
////////////////////////////////////////////////////////////////////////
{
//    Int32 i = BUFFER_COUNT * NUM_BUFFERS;
       Int32 i = BUFFER_LENGTH * NUM_BUFFERS;
    Int32 *p = (Int32 *)buffer;

    while(i--)
        *p++ = 0;


    init_W(N, W); // initialize fft twiddle factors
    init_W_neg(N, NW); // initialize ifft twiddle factors
```

```
    padTo1024(B, N_coeff);

    for(i = 0; i < N; i++)
    {
       coeffs[i].real = B_pad[i];
       coeffs[i].imag = 0.0f;
    }

    for(i = 0; i < (N - BUFFER_COUNT); i++) {
//     saved[i].real = 0;
//     saved[i].imag = 0;
       saved[i] = 0;
    }

    fft_c(N, coeffs, W);
}

void ProcessBuffer()
////////////////////////////////////////////////////////////////////////
// Purpose:   Processes the data in buffer[ready_index] and stores
//              the results back into the buffer
//            Data is packed into the buffer, alternating right/left
//
// Input:     None
//
// Returns:   Nothing
//
// Calls:     Nothing
//
// Notes:     None
////////////////////////////////////////////////////////////////////////
{
       Int16 *pBuf = buffer[ready_index];
       Int32 i;
    float tmp1, tmp2;

       /* copy right channel to complex array */
   for(i=0;i < BUFFER_COUNT;i++){
              x[i].real = *pBuf;
              x[i].imag = 0;
              pBuf += 2; // skip left channel
   }

   for(i = BUFFER_COUNT; i < N; i++){
          x[i].real = 0;
          x[i].imag = 0;
   }

       pBuf = buffer[ready_index];
       /* calculate the FFT of x[N] */
       fft_c(N, x, W);

       for(i = 0; i < N; i++)
       {
              tmp1 = x[i].real * coeffs[i].real;
              tmp2 = x[i].imag * coeffs[i].imag;
              mult_result[i].real = tmp1 - tmp2;
```

```
              mult_result[i].imag = (x[i].real + x[i].imag)*(coeffs[i].real +
coeffs[i].imag) - tmp1 - tmp2;
       }

       fft_c(N, mult_result, NW);

   for(i=0;i < N;i++){
          if( i < (N - BUFFER_COUNT)) {
                 tmp1 = ( mult_result[i].real / N ) + saved[i];
          }
          else if( i >= BUFFER_COUNT )
          {
                 tmp1 = ( mult_result[i].real / N );
                 saved[i - BUFFER_COUNT] = tmp1;
          }
          else
          {
                 tmp1 = ( mult_result[i].real / N );
          }
          if( i < BUFFER_COUNT ) {
                 *pBuf++ = _spint(tmp1 * 65536) >> 16;
                 *pBuf++ = 0;
          }
   }

       pBuf = buffer[ready_index];

    buffer_ready = 0; // signal we are done
}

//////////////////////////////////////////////////////////////////////
// Purpose:   Access function for buffer ready flag
//
// Input:     None
//
// Returns:   Non-zero when a buffer is ready for processing
//
// Calls:     Nothing
//
// Notes:     None
//////////////////////////////////////////////////////////////////////
int IsBufferReady()
{
    return buffer_ready;
}

//////////////////////////////////////////////////////////////////////
// Purpose:   Access function for buffer overrun flag
//
// Input:     None
//
// Returns:   Non-zero if a buffer overrun has occurred
//
// Calls:     Nothing
//
// Notes:     None
//////////////////////////////////////////////////////////////////////
int IsOverRun()
```

```
{
    return over_run;
}

interrupt void EDMA_ISR()
////////////////////////////////////////////////////////////////////////
// Purpose:   EDMA interrupt service routine.  Invoked on every buffer
//            completion
//
// Input:     None
//
// Returns:   Nothing
//
// Calls:     Nothing
//
// Notes:     None
////////////////////////////////////////////////////////////////////////
{
       *(volatile Uint32 *)CIPR = 0xf000; // clear all McBSP events
    if(++ready_index >= NUM_BUFFERS) // update buffer index
        ready_index = 0;
    if(buffer_ready == 1) // set a flag if buffer isn't processed in time
        over_run = 1;
    buffer_ready = 1; // mark buffer as ready for processing
}

void padTo1024(float B_arg[], int bSize )
{
       int i;
       for(i = 0; i < bSize; i++)
              B_pad[i] = B_arg[i];

       for(i = bSize; i < N; i++)
              B_pad[i] = 0.0f;

}
```