Real-Time Fast Fourier Transform

10

Introduction

The Fourier transform is a standard system analysis tool for viewing the spectral content of a signal or sequence. The Fourier transform of a sequence, commonly referred to as the discrete time Fourier transform or DTFT is not suitable for real-time implementation. The DTFT takes a sequence as input, but produces a continuous function of frequency as output. A close relative to the DTFT is the discrete Fourier transform or DFT. The DFT takes a finite length sequence as input and produces a finite length sequence as output. When the DFT is implemented as an efficient algorithm it is called the Fast Fourier Transform (FFT). The FFT is ultimately the subject of this chapter, as the FFT lends itself to real-time implementation. The most popular FFT algorithms are the radix 2 and radix 4, in either a decimation in time or a decimation in frequency signal flow graph form (transposes of each other)

DTFT, DFT, and FFT Theory Overview

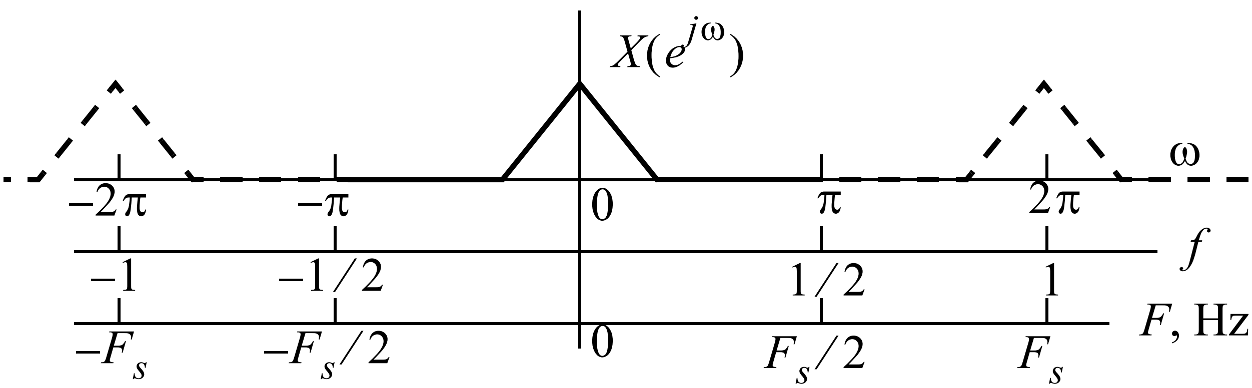
Frequency domain analysis of discrete-time signals begins with the DTFT, but for practicality reasons moves to the DFT and then to the more efficient FFT.

The DTFT

- Recall that the DTFT of a sequence $x[n]$ is simply the sum

$$X(e^{j\omega}) = X(\omega) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n} \tag{10.1}$$

- A characteristic of the spectrum $X(e^{j\omega})$ is that it is a periodic function having period 2π
- The frequency axis in (10.1) is in radians per sample, which is a normalized frequency variable
- If $x[n]$ was acquired by sampling an analog signal every T seconds, e.g., $x[n] = x_a(nT)$, then the ω frequency axes is related to the normalized digital frequency $f = \omega/(2\pi)$ and the analog frequency $F = f \cdot F_s$ as follows:



The DFT and its Relation to the DTFT

- The DFT assumes that the sequence $y[n]$ exists only on a finite interval, say $0 \le n \le N-1$, and zero otherwise
- This may occur as a result of data windowing, e.g.,

$$y[n] = x[n]w[n] \tag{10.2}$$

where $w[n]$ is a window function of duration N samples and zero otherwise

– Unless defined otherwise $w[n]$ is a rectangular window, but it may be a shaped window of the same type as used in windowed FIR filter design

- The DTFT of $y[n]$ is

$$Y(e^{j\omega}) = \sum_{n=0}^{N-1} y[n]e^{-j\omega n}, \forall\omega \tag{10.3}$$

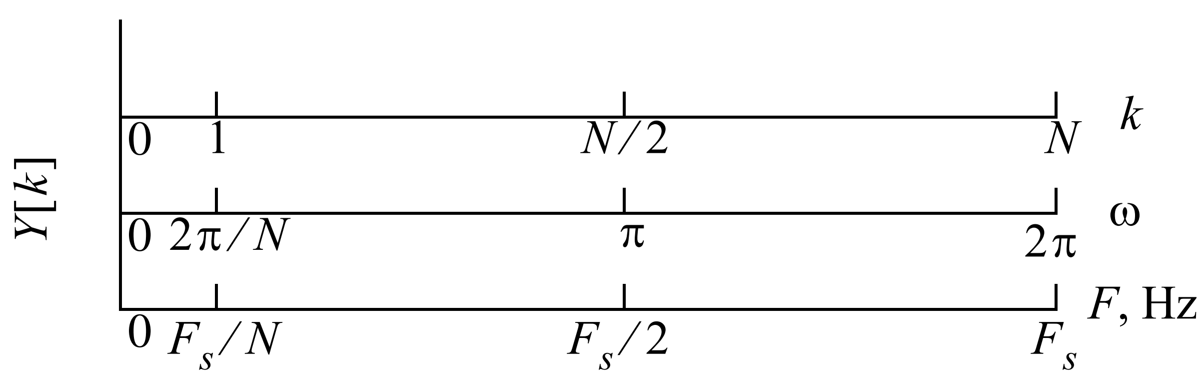
- The DFT of $y[n]$ is defined to be

$$Y[k] = \sum_{n=0}^{N-1} y[n]e^{-\frac{j2\pi kn}{N}}, k = 0, 1, \ldots, N-1 \tag{10.4}$$

- The relation between (10.3) and (10.4) is simply that

$$Y[k] = Y(e^{j\omega})\Big|_{\omega = \frac{2\pi k}{N}} \tag{10.5}$$

- DFT frequency axis relationships are the following:



- A feature of $Y[k]$ is that it is computable; about N^2 complex multiplications and $N(N-1)$ complex additions are required for an N-point DFT, while the radix-2 FFT, discussed shortly, reduces the complex multiplication count down to about $(N/2)\log_2 N$
- The inverse DFT or IDFT is defined to be

$$y[n] = \frac{1}{N}\sum_{k=0}^{N-1} Y[k]e^{\frac{j2\pi kn}{N}}, n = 0, 1, \ldots, N-1 \tag{10.6}$$

- In both the DFT and IDFT expressions it is customary to let

$$W_N \equiv e^{-j\frac{2\pi}{N}} \tag{10.7}$$

then the DFT and IDFT pair can be written as

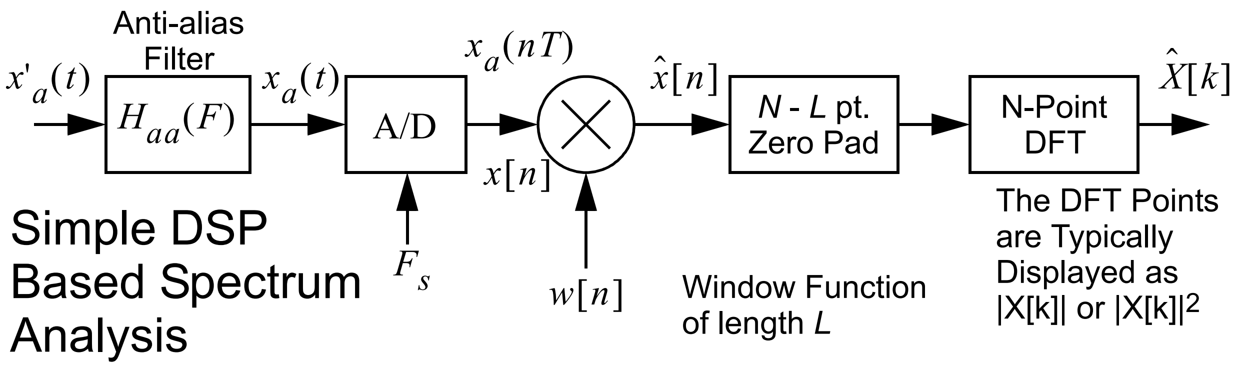
$$\boxed{\begin{aligned} Y[k] &= \sum_{n=0}^{N-1} y[n]W_N^{kn}, k = 0, 1, \ldots, N-1 \\ y[n] &= \frac{1}{N}\sum_{k=0}^{N-1} Y[k]W_N^{-kn}, n = 0, 1, \ldots, N-1 \end{aligned}} \tag{10.8}$$

  – The complex weights W_N^k are known as *twiddle constants* or factors

Simple Application Examples

To further motivate study of the FFT for real-time applications, we will briefly consider two spectral domain processing examples

Spectral Analysis



- Suppose that

$$x_a(t) = A_1 \cos(2\pi F_1 t) + A_2 \cos(2\pi F_2 t) \tag{10.9}$$

and $F_1, F_2 \leq F_s/2$

- The true spectrum, assuming an infinite observation time, is

$$\begin{aligned} X(\omega) = {} & \frac{A_1}{2}[\delta(\omega - \omega_1) + \delta(\omega + \omega_1)] \\ & + \frac{A_2}{2}[\delta(\omega - \omega_2) + \delta(\omega + \omega_2)] \end{aligned} \tag{10.10}$$

for $-\pi < \omega \leq \pi$ and $\omega_1 = 2\pi F_1/F_s$ and $\omega_2 = 2\pi F_2/F_s$

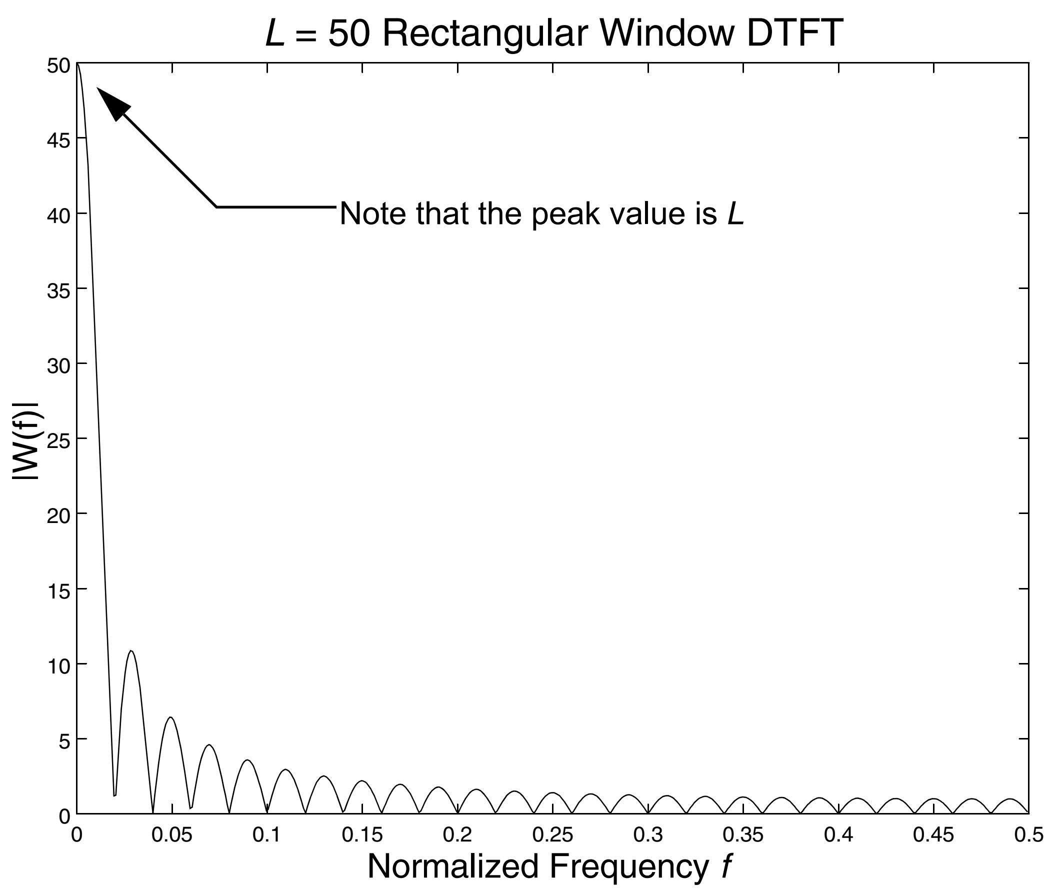
- As a result of windowing the data

$$\hat{X}(\omega) = \frac{A_1}{2}[W(\omega - \omega_1) + W(\omega + \omega_1)] + \frac{A_2}{2}[W(\omega - \omega_2) + W(\omega + \omega_2)] \tag{10.11}$$

for $-\pi < \omega \leq \pi$, where $W(\omega) = \text{DTFT}\{w[n]\}$

  – A rectangular window (in MATLAB `boxcar`) of length L samples has DTFT

$$W(\omega) = \frac{\sin(\omega L/2)}{\sin(\omega/2)} e^{-j\omega(L-1)/2} \tag{10.12}$$



- The DFT points, $\hat{X}[k]$, are simply a sampled version of (10.11), that is,

$$\begin{aligned}\hat{X}[k] = \frac{A_1}{2}\Big[W\Big(\frac{2\pi k}{N} - \omega_1\Big) + W\Big(\frac{2\pi k}{N} + \omega_1\Big)\Big] \\ + \frac{A_2}{2}\Big[W\Big(\frac{2\pi k}{N} - \omega_2\Big) + W\Big(\frac{2\pi k}{N} + \omega_2\Big)\Big]\end{aligned} \tag{10.13}$$

  for $0 \leq k \leq N$

- The zero-padding operation is optional, but is often used to improve the apparent spectral resolution
- Without zero padding we have only L samples on $(0, 2\pi)$, while with zero padding we effectively make the DFT create additional samples of $X(\omega)$, raising the number from L to N
- The frequency sample spacing changes from

$$\Delta\omega = \frac{2\pi}{L} \text{ to } \Delta\omega = \frac{2\pi}{N} \text{ radians/sample}$$

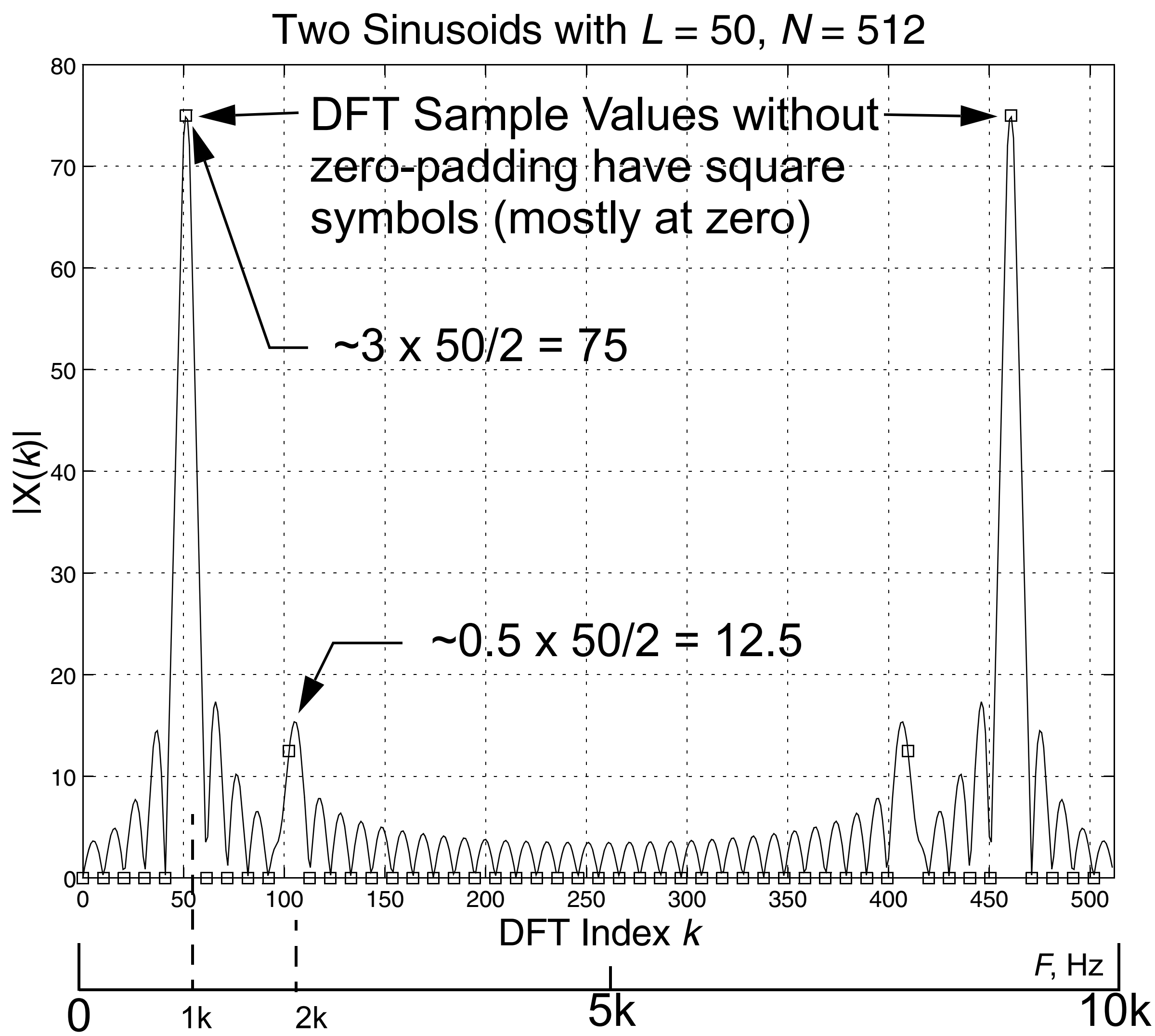
  or

$$\Delta f = \frac{1}{L} \text{ to } \Delta f = \frac{1}{N} \text{ cycles/sample}$$

Example:

```
» n = 0:49;
» x = 3*cos(2*pi*1000*n/10000) + ...
  0.5*cos(2*pi*2000*n/10000); % Fs = 10000 Hz
» X = fft(x,512);
» plot(0:511,abs(X))
» grid
```

```
» title('Two Sinusoids with L = 50','fontsize',18)
» ylabel('|X(k)|','fontsize',16)
» xlabel('DFT Index k','fontsize',16)
» axis([0 512 0 80]);
» printmif('5910_2')
```

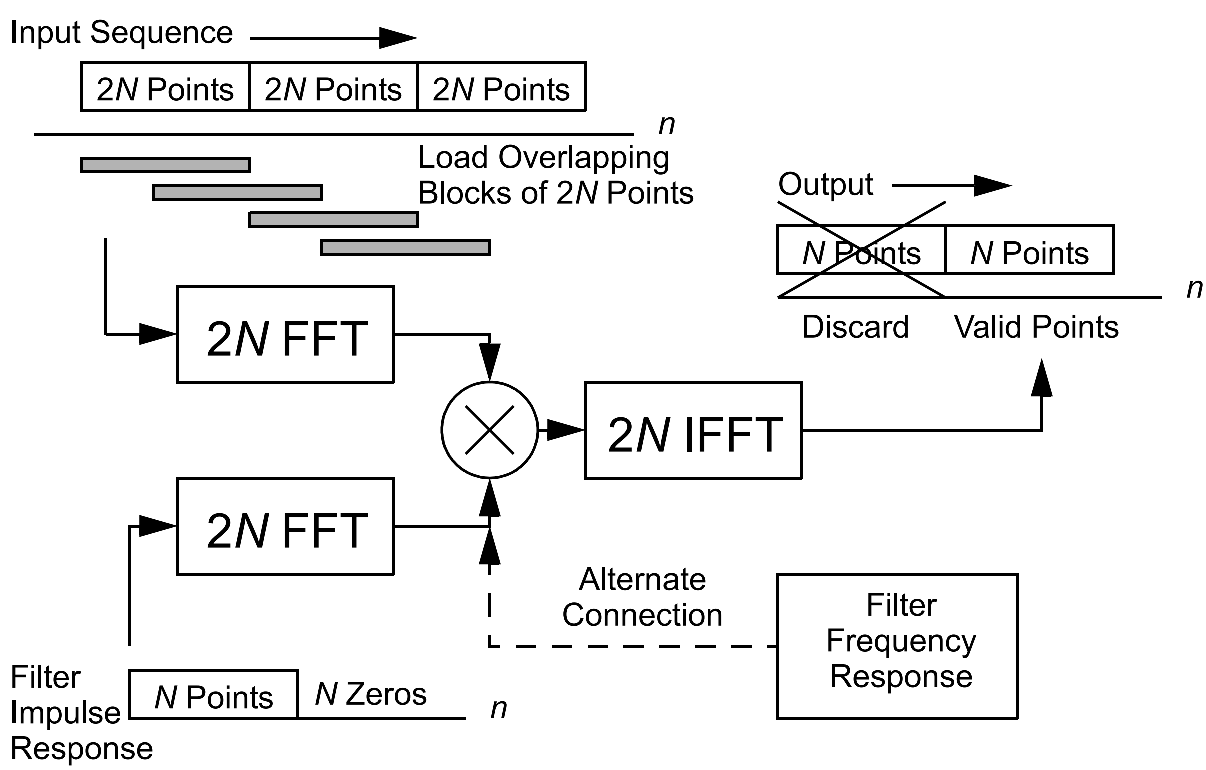


- With spectral analysis the goal may not be to actually display the spectrum, but rather to perform some additional computations on the DFT points

- For example we may try to decide if the frequency domain features of the transformed signal record are close to some template, i.e., target detection, pattern recognition/classification, etc.

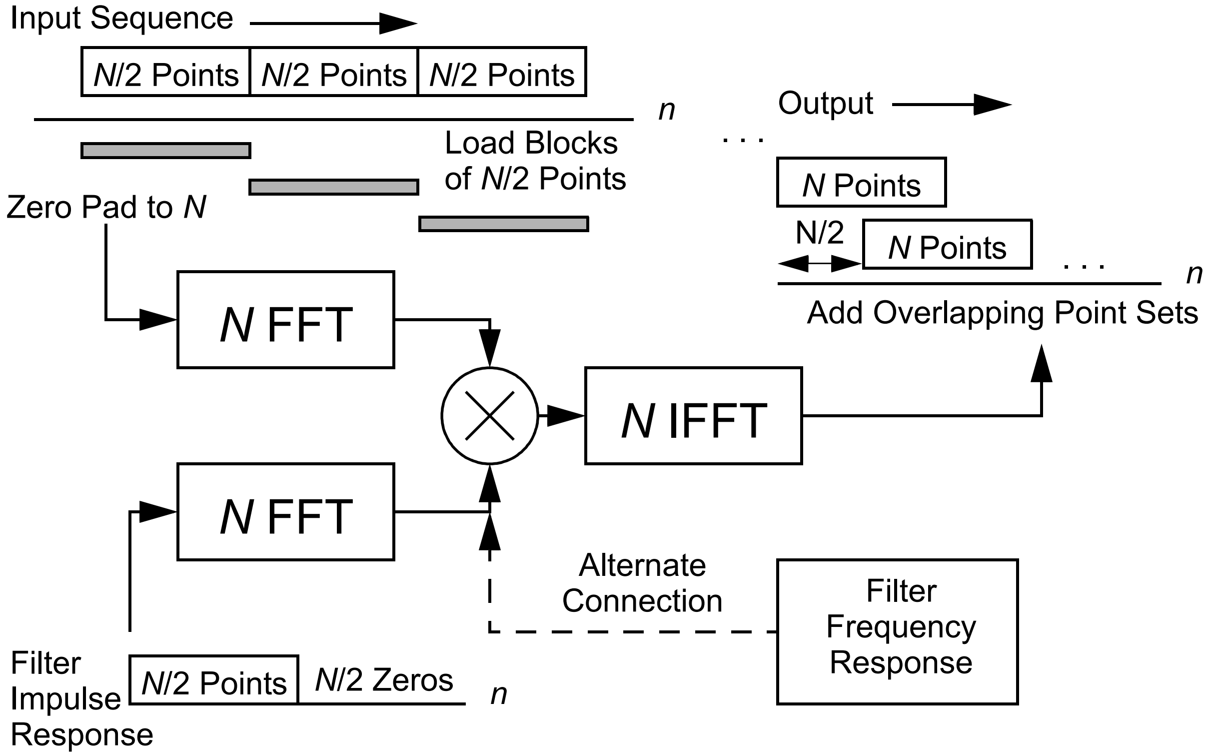
Transform Domain Linear Filtering

- A considerably different application of the DFT is to perform filtering directly in the frequency domain
- The basic assumption here is that the signal sample stream we wish to filter may have infinite duration, but the impulse response of the filter must be finite
- One standard approach is known as the *overlap and save* method of FIR filtering



Overlap and Save Filtering

- As the impulse response length grows the transform domain approach becomes more efficient than the time domain convolution sum, implemented in say a direct form FIR structure
- Another way of performing a transform domain convolution is with the *overlap and add* method



Overlap and Add Filtering

Radix 2 FFT

- A divide-and-conquer approach is taken in which an N-point (N a power of two, $N = 2^{\nu}$) DFT is decomposed into a cascade like connection of 2-point DFTs
  - With the decimation-in-time (DIT) algorithm the decomposition begins by decimating the input sequence $x[n]$

– With the decimation-in-frequency (DIF) algorithm the decomposition begins by decimating the output sequence $X[k]$ and working backwards

Decimation-in-Time

- Begin by separating $x[n]$ into even and odd $N/2$ point sequences

$$X[k] = \sum_{n \text{ even}} x[n] W_N^{kn} + \sum_{n \text{ odd}} x[n] W_N^{kn} \tag{10.14}$$

- In the first sum let $n = 2r$ and in the second let $n = 2r + 1$

$$X[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r] W_N^{2rk} + \sum_{r=0}^{\frac{N}{2}-1} x[2r+1] W_N^{(2r+1)k} \tag{10.15}$$

or

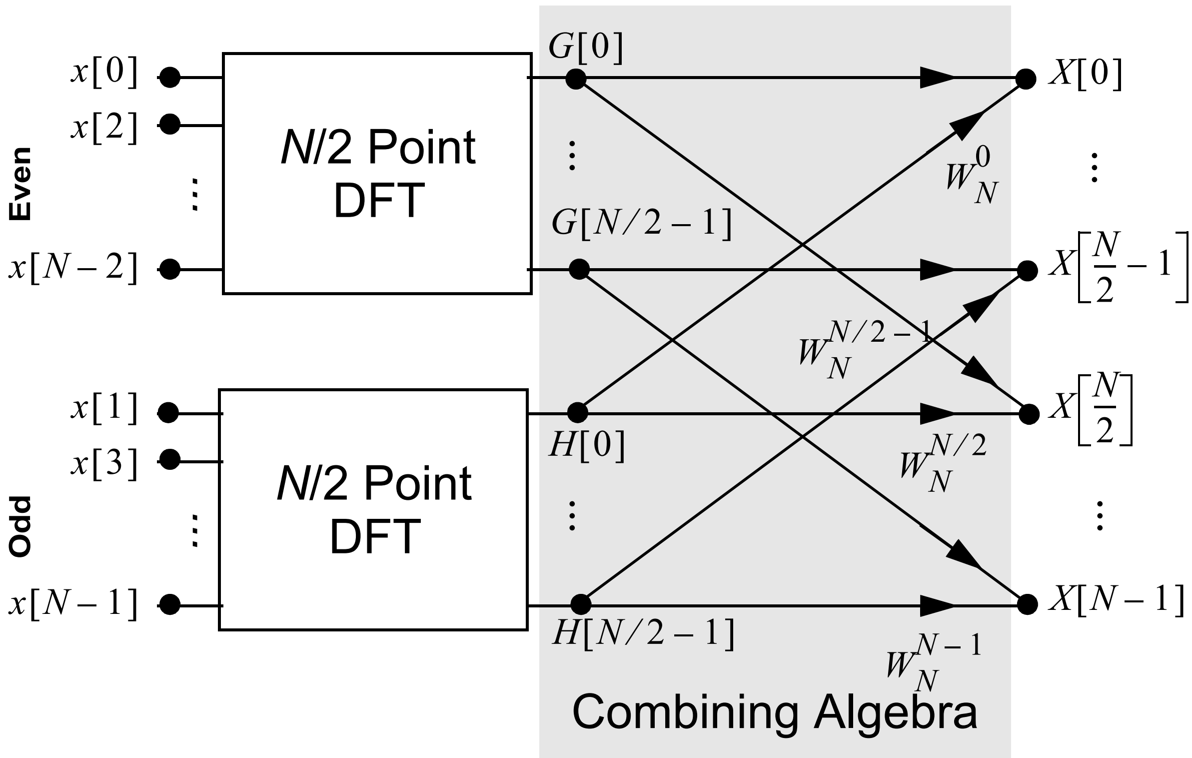
$$X[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r] (W_N^2)^{rk} + W_N^k \sum_{r=0}^{\frac{N}{2}-1} x[2r+1] (W_N^2)^{rk} \tag{10.16}$$

- Note that $W_N^2 = e^{-2j(2\pi/N)} = e^{-j2\pi/(N/2)} = W_{N/2}$, thus

$$\begin{aligned} X[k] &= \sum_{r=0}^{\frac{N}{2}-1} x[2r] W_{N/2}^{rk} + W_N^k \sum_{r=0}^{\frac{N}{2}-1} x[2r+1] W_{N/2}^{rk} \qquad (10.17) \\ &= G[k] + W_N^k H[k] \end{aligned}$$

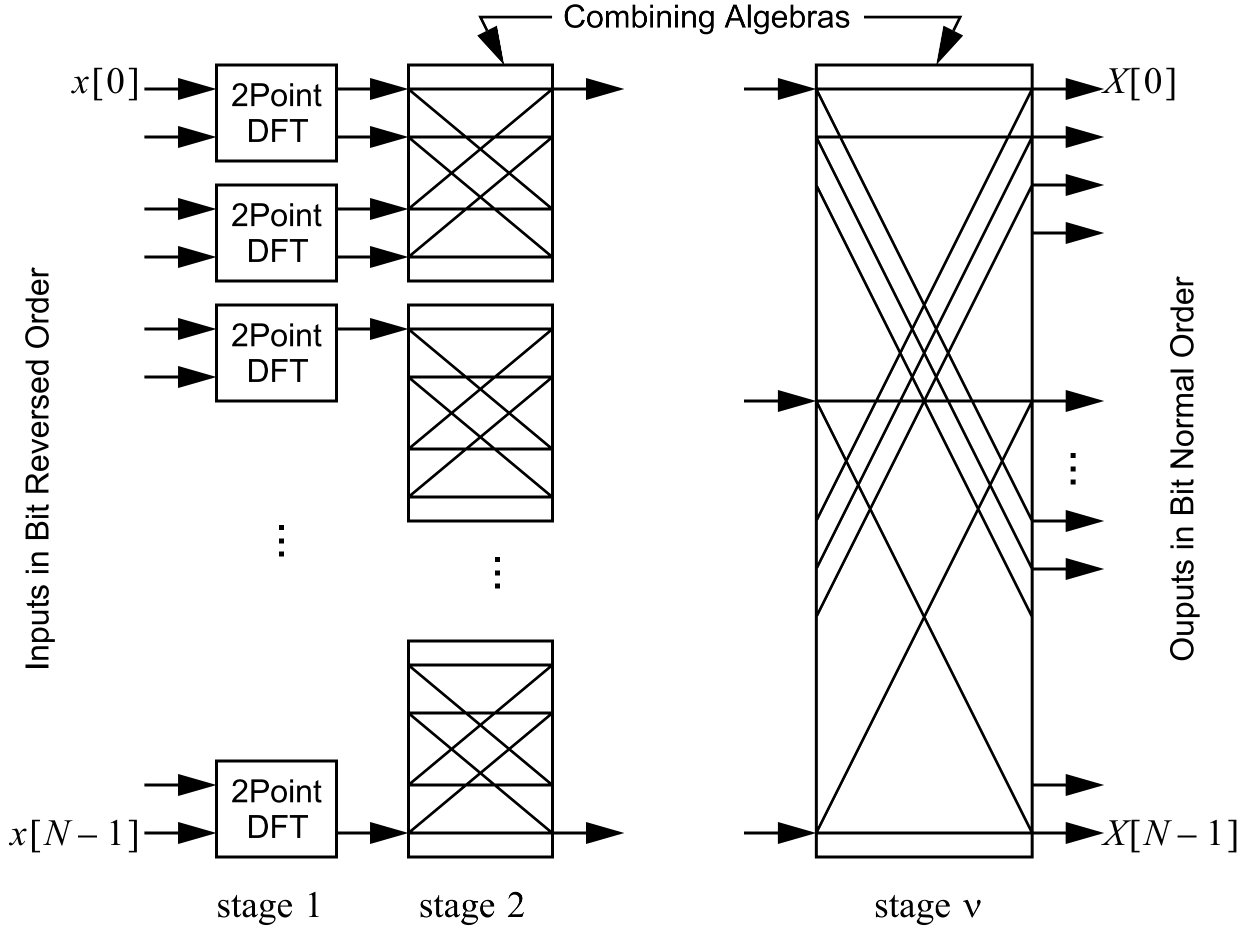
where we now see that $G[k]$ and $H[k]$ are $N/2$ point DFTs of the even and odd points of the original sequence $x[n]$

- A *combining* algebra brings the $N/2$-point DFTs together

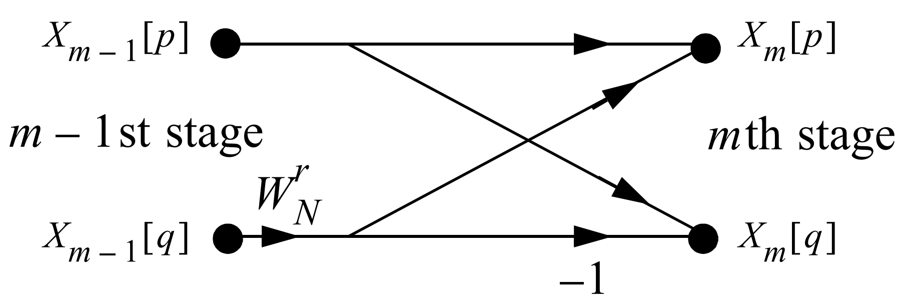


- After this one stage of decimation a computational savings has already been achieved
  - The number of complex multiplies is $2(N/2)^2 + N = N^2/2 + N \leq N^2$ for $N \geq 2$
- The procedure can be repeated until there are a total of $\nu = \log_2 N$ stages of decimation

- The general form of the DIT FFT signal flow graph is now



- When the decimation is complete the basic computation in each of the two point DFTs and the combining algebras all look the same
- We have what is known as the *radix-2 butterfly*



Simplified Radix-2 Butterfly

- The number stages is $\log_2 N$ and when using the simplified butterfly one complex multiply is required per stage with $N/2$ butterflies per stage, thus the total number of complex multiplies per FFT is

$$\text{Multiplications} = \frac{N}{2}\log_2 N \tag{10.18}$$

  – Note: Each complex multiply requires four real multiplies and two real additions

- The number of complex additions is

$$\text{Additions} = N\log_2 N \tag{10.19}$$

  – Note: Each complex addition requires two real additions

- Further study of the DIT algorithm reveals that it can be computed *in-place*, that is only one complex array is required to transform $x[n]$ into $X[k]$

- A drawback is that the input array must be initially reordered into *bit reversed* order, that is say for $N = 8$

$$
\begin{aligned}
x[0] &= x[000] \Rightarrow X[000] = X[0] \\
x[4] &= x[100] \Rightarrow X[001] = X[1] \\
x[2] &= x[010] \Rightarrow X[010] = X[2] \\
&\cdots \\
x[3] &= x[011] \Rightarrow X[110] = X[6] \\
x[7] &= x[111] \Rightarrow X[111] = X[7]
\end{aligned}
$$

Decimation-in-Frequency

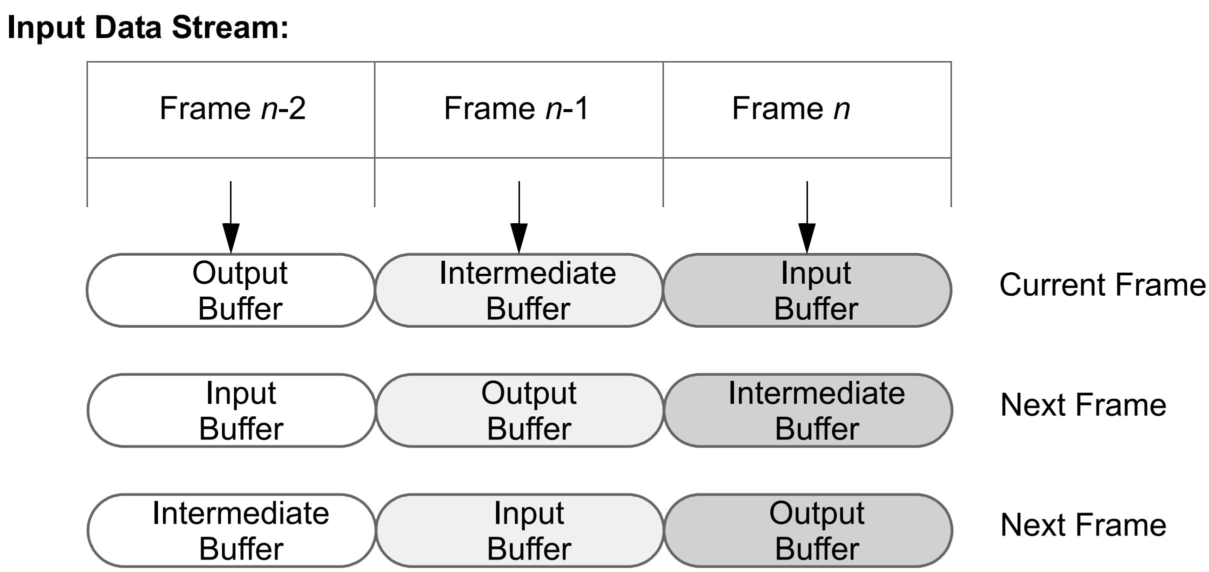
- With the DIF approach the decimation starts at the output and works towards the input
- The signal flow graph turns out to be the transposed version of the DIT flow graph
  - The in-place calculation property holds again
  - The operations count is identical
  - A significant difference is that the input is in normal order and the output is now in bit reversed order

Computing the IDFT

- To compute the IDFT we do it the same way as the DFT, except we must include a $1/N$ scale factor and the twiddle constants must be conjugated

Frame Processing

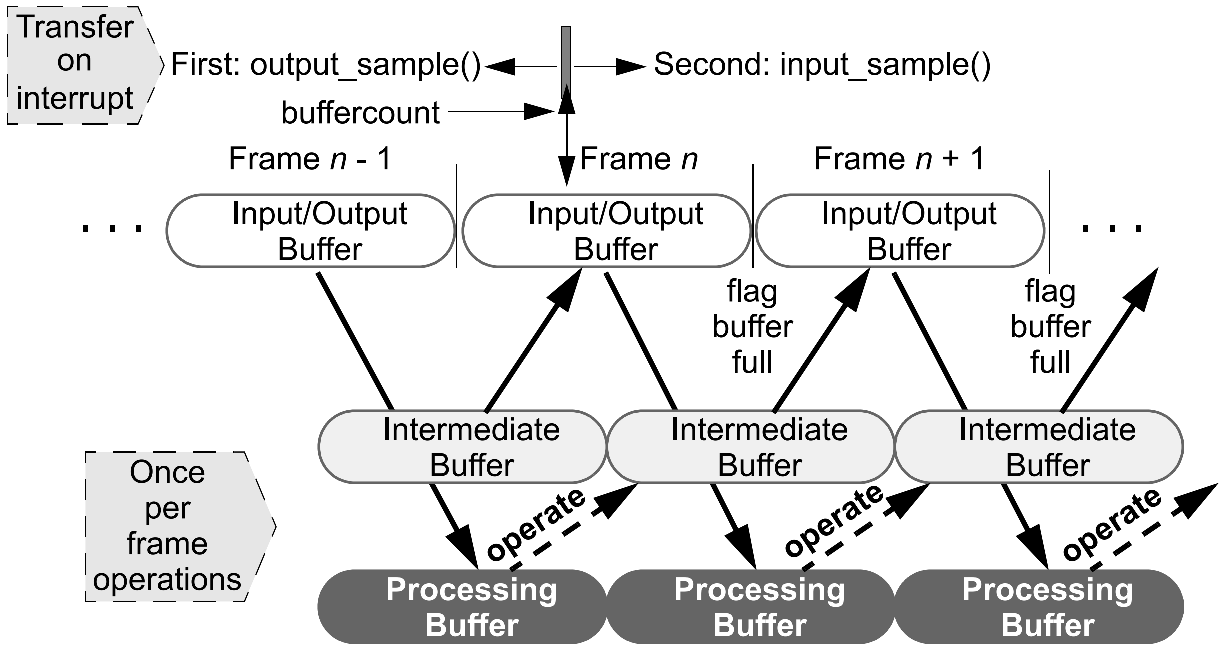
- The FFT is inherently a frame-by-frame or block oriented algorithm
- To compute a real-time FFT using streaming input data, we typically need to set up a buffer system
- One approach is the triple buffer system shown below:



  – The buffers are just blocks of memory

- As the *Input buffer* is filling sample-by-sample from the A/D, the FFT is being computed using the *Intermediate Buffer* as a complete frame of data
- At the same time the output buffer is writing its contents out sample-by-sample to the D/A
- Note that with this scheme there is an inherent two frame delay or lag

- A variation on the above scheme is to use a single input/output buffer, one intermediate buffer, and a primary buffer



- In this scheme each of the three buffers assumes the same role on each frame cycle, while in the previous scheme the roles changed periodically (three frame period)

Radix-2 FFT Implementation on the C6x

Sample code is provided in the text using an entirely C based FFT and using linear assembly files from a TI code library. The C code is the easiest to use, but is not as efficient as the .sa files.

A C Based FFT for Real-Time Spectrum Analysis Using and Oscilloscope

- This example is Chassaing's `fft128c.c` program (Example 6.5, p. 290)

- The three buffer approach is used in this program
- The buffers are declared as `A[N]`, `B[N]`, and `C[N]`, but the roles they serve is controlled by revolving float pointers, `*input_ptr`, `*process_ptr`, and `*output_ptr`; a temporary pointer, `*temp_ptr` is also utilized

```
//fft128c.c

#include "DSK6713_AIC23.h"//codec-DSK interface support
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;//set sampling rate
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource=DSK6713_AIC23_INPUT_LINE;

#include <math.h>
#include "fft.h"
#define PI 3.14159265358979
#define TRIGGER 32000
#define N 128
#include "hamm128.h"

short buffercount = 0;    //number of new input samples in iobuffer
short bufferfull = 0;     //set by ISR to indicate iobuffer full
COMPLEX A[N], B[N], C[N];
COMPLEX *input_ptr, *output_ptr, *process_ptr, *temp_ptr;
COMPLEX twiddle[N];
short outbuffer[N];

interrupt void c_int11(void)      //ISR
{
  output_left_sample((short)((output_ptr + buffercount)->real));
  outbuffer[buffercount] =
                  -(short)((output_ptr + buffercount)->real);
  (input_ptr + buffercount)->real = (float)(input_left_sample());
  (input_ptr + buffercount++)->imag = 0.0;
  if (buffercount >= N)       //for overlap-add method iobuffer
  {                             // is half size of FFT used
    buffercount = 0;
```

```
    bufferfull = 1;
  }
}

main()
{
  int n;

  for (n=0 ; n<N ; n++)         //set up DFT twiddle factors
  {
    twiddle[n].real = cos(PI*n/N);
    twiddle[n].imag = -sin(PI*n/N);
  }
  input_ptr = A;
  output_ptr = B;
  process_ptr = C;
  comm_intr();                    //initialise DSK, codec, McBSP
  while(1)                        //frame processing loop
  {
    while(bufferfull==0);    //wait for new frame of input samples
    bufferfull = 0;

    temp_ptr = process_ptr; //rotate buffer/frame pointers
    process_ptr = input_ptr;
    input_ptr = output_ptr;
    output_ptr = temp_ptr;

    fft(process_ptr,N,twiddle); //process contents of buffer

    for (n=0 ; n<N ; n++)       // compute magnitude of frequency
                                //domain representation
    {                           // and place in real part
      (process_ptr+n)->real = -sqrt((process_ptr+n)->real*
                                  (process_ptr+n)->real
                              + (process_ptr+n)->imag*
                                (process_ptr+n)->imag)/16.0;
    }
    (process_ptr)->real = TRIGGER; // add scope trigger pulse
  }                                  //end of while(1)
}                                    //end of main()
```

- The C-based fft is a decimation frequency algorithm, by virtue of the fact that the reordering operation is done at the end

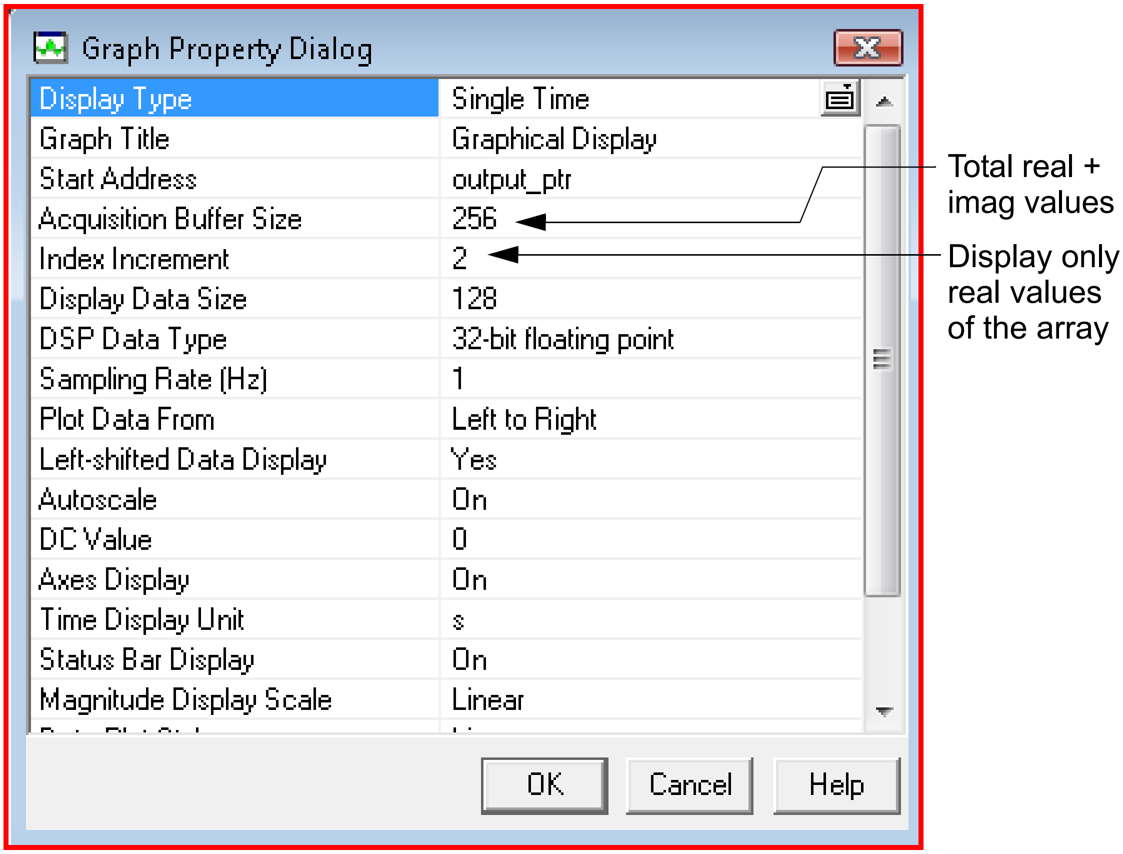
```
//fft.h complex FFT function taken from Rulph's C31 book
//this file contains definition of complex data structure also

struct cmpx                       //complex data structure used by
FFT
    {
    float real;
    float imag;
    };
typedef struct cmpx COMPLEX;

void fft(COMPLEX *Y, int M, COMPLEX *w)  //input sample array,
                                         //number of points
{
  COMPLEX temp1,temp2;      //temporary storage variables
  int i,j,k;                //loop counter variables
  int upper_leg, lower_leg; //index of upper/lower butterfly leg
  int leg_diff;             //difference between upper/lower leg
  int num_stages=0;         //number of FFT stages, or iterations
  int index, step;         //index and step between twiddle factor
  i=1;                 //log(base 2) of # of points = # of stages
  do
  {
    num_stages+=1;
    i=i*2;
  } while (i!=M);

  leg_diff=M/2;   //starting difference between upper & lower legs
  step=2;         //step between values in twiddle.h
  for (i=0;i<num_stages;i++)      //for M-point FFT
  {
    index=0;
    for (j=0;j<leg_diff;j++)
    {
      for (upper_leg=j;upper_leg<M;upper_leg+=(2*leg_diff))
      {
```

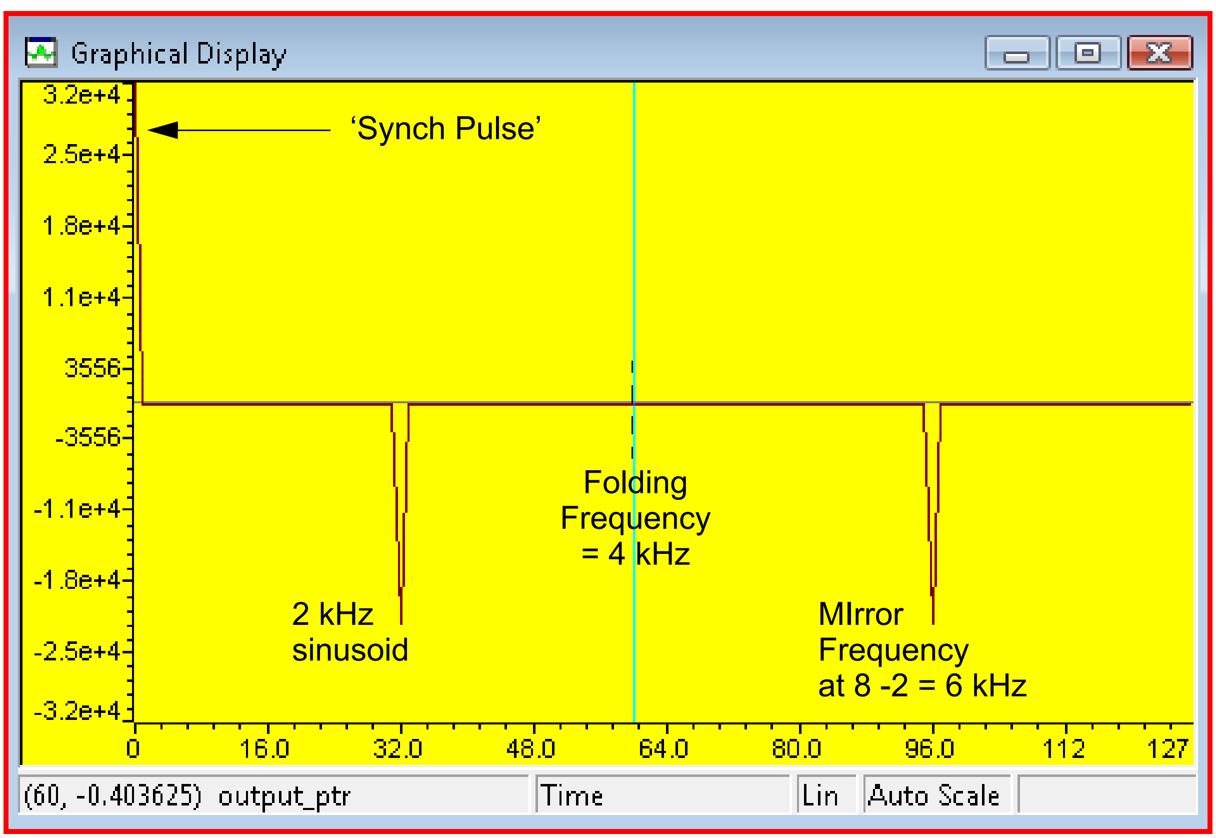
```
        lower_leg=upper_leg+leg_diff;
        temp1.real=(Y[upper_leg]).real + (Y[lower_leg]).real;
        temp1.imag=(Y[upper_leg]).imag + (Y[lower_leg]).imag;
        temp2.real=(Y[upper_leg]).real - (Y[lower_leg]).real;
        temp2.imag=(Y[upper_leg]).imag - (Y[lower_leg]).imag;
        (Y[lower_leg]).real=temp2.real*(w[index]).real-
temp2.imag*(w[index]).imag;
        (Y[lower_leg]).imag=temp2.real*(w[index]).imag+
                            temp2.imag*(w[index]).real;
        (Y[upper_leg]).real=temp1.real;
        (Y[upper_leg]).imag=temp1.imag;
      }
      index+=step;
    }
    leg_diff=leg_diff/2;
    step*=2;
  }
  j=0;
  for (i=1;i<(M-1);i++)    //bit reversal for resequencing data
  {
    k=M/2;
    while (k<=j)
    {
      j=j-k;
      k=k/2;
    }
    j=j+k;
    if (i<j)
    {
      temp1.real=(Y[j]).real;
      temp1.imag=(Y[j]).imag;
      (Y[j]).real=(Y[i]).real;
      (Y[j]).imag=(Y[i]).imag;
      (Y[i]).real=temp1.real;
      (Y[i]).imag=temp1.imag;
    }
  }
  return;
}              //end of fft()
```

- Audio samples are stored in the array pointed to by `input_ptr`
- Since the FFT operates on complex signal samples, a complex structure variable `COMPLEX` is defined to hold two `floats`, `real` and `imag`
- The intermediate buffer are stored in the array pointed to by `process_ptr`
- Twiddle factors used in the FFT are stored in the global `COMPLEX` vector `twiddle[N]`
- Once the main C routine returns from `fft()`, the transformed values at `process_ptr` are manipulated to form

  $$|X[k]| = \sqrt{X_R^2[k] + X_I^2[k]},$$

  and are ultimately show up in `output_ptr` when output to the D/A
- A negative spike for oscillocope sync is also inserted at dc (actually inverted in the ISR when writing to the codec)
- If we use CCS to plot `output_ptr`, with an array stride factor of 2, we will see a combination of time and frequency domain data


Graph Property Dialog
Display Type
Single Time
Graph Title
Graphical Display
Start Address
output_ptr
Acquisition Buffer Size
256
Index Increment
2
Display Data Size
128
DSP Data Type
32-bit floating point
Sampling Rate (Hz)
1
Plot Data From
Left to Right
Left-shifted Data Display
Yes
Autoscale
On
DC Value
0
Axes Display
On
Time Display Unit
s
Status Bar Display
On
Magnitude Display Scale
Linear
OK
Cancel
Help
Total real + imag values
Display only real values of the array


Graphical Display
3.2e+4
2.5e+4
1.8e+4
1.1e+4
3556
-3556
-1.1e+4
-1.8e+4
-2.5e+4
-3.2e+4
0
16.0
32.0
48.0
64.0
80.0
96.0
112
127
'Synch Pulse'
Folding Frequency = 4 kHz
2 kHz sinusoid
MIrror Frequency at 8 -2 = 6 kHz
(60, -0.403625) output_ptr
Time
Lin
Auto Scale

Transform Domain FIR Filtering

- This program implements the overlap-and-save method of fast convolution for FIR filtering (Example 6.12, pp. 308ff)
- The same triple buffer configuration as the previous example is employed
- The FFT is also identical
-

```
//fastconv.c

#include "DSK6713_AIC23.h"//codec-DSK interface support
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;//set sampling rate
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource=DSK6713_AIC23_INPUT_LINE; // select input
source

#include "lp55f.cof"

#include <math.h>
#include "fft.h"
#define PI 3.14159265358979
#define PTS 128

short buffercount = 0;    // index into frames
short bufferfull=0;
COMPLEX A[PTS], B[PTS], C[PTS];   //three buffers used
COMPLEX twiddle[PTS];              //twiddle factors
COMPLEX coeffs[PTS];              //zero padded freq domain filter
coeffs
//pointers to frames
COMPLEX *input_ptr, *output_ptr, *process_ptr
float a,b;          //variables used in complex multiply

interrupt void c_int11(void)      //ISR
{
```

```
  output_left_sample((short)((output_ptr + buffercount)->real));
  (input_ptr + buffercount)->real = (float)(input_left_sample());
  (input_ptr + buffercount++)->imag = 0.0;
  if (buffercount >= PTS/2)
  {
    bufferfull = 1;
    buffercount = 0;
  }
}

void main()
{
  int n,i;
  for (n=0 ; n<PTS ; n++)  //set up twiddle factors in array w
  {
    twiddle[n].real = cos(PI*n/PTS);
    twiddle[n].imag = -sin(PI*n/PTS);
  }
  for (n=0 ; n<PTS ; n++)//set up complex freq domain filt coeffs
  {
    coeffs[n].real = 0.0;
    coeffs[n].imag = 0.0;
  }
  for (n=0 ; n<N ; n++)
  {
    coeffs[n].real = h[n];
  }
  fft(coeffs,PTS,twiddle);//transform filter coeffs to freq domain
  input_ptr = A;  //initialise pointers to frames/buffers
  process_ptr = B;
  output_ptr = C;
  comm_intr();


  while(1)                        //frame processing loop
  {
    while (bufferfull == 0);            //wait for iobuffer full
    bufferfull = 0;
    temp_ptr = process_ptr;
    process_ptr = input_ptr;
```

```
    input_ptr = output_ptr;
    output_ptr = temp_ptr;

    for (i=0 ; i< PTS ; i++) (process_ptr + i)->imag = 0.0;
    for (i=PTS/2 ; i< PTS ; i++) (process_ptr + i)->real = 0.0;

    fft(process_ptr,PTS,twiddle); //transform samples into
                                  //the frequency domain

    for (i=0 ; i<PTS ; i++)      //filter frequency domain
                                 //representation
    {                //i.e. complex multiply samples by coeffs
      a = (process_ptr + i)->real;
      b = (process_ptr + i)->imag;
      (process_ptr + i)->real = coeffs[i].real*a -
                                coeffs[i].imag*b;
      (process_ptr + i)->imag = -(coeffs[i].real*b +
                                  coeffs[i].imag*a);
    }
    fft(process_ptr,PTS,twiddle);
    for (i=0 ; i<PTS ; i++)
    {
      (process_ptr + i)->real /= PTS;
      //if result is real, do we need this?
      (process_ptr + i)->imag /= -PTS;
    }
    for (i=0 ; i<PTS/2 ; i++) //overlap add (real part only!!)
    {
      (process_ptr + i)->real += (output_ptr + i + PTS/2)->real;
    }

  } // end of while
}                                 //end of main()
```

- The filter in this case is lowpass, having coefficients

```
//lp55.cof low pass filter

#define N 55

float h[N] = {
```

```
 3.6353E-003,-1.1901E-003,-4.5219E-003, 2.6882E-003, 5.1775E-003,
-4.7252E-003,-5.4097E-003, 7.2940E-003, 4.9986E-003,-1.0343E-002,
-3.6979E-003, 1.3778E-002, 1.2276E-003,-1.7460E-002, 2.7529E-003,
 2.1222E-002,-8.7185E-003,-2.4870E-002, 1.7465E-002, 2.8205E-002,
-3.0642E-002,-3.1035E-002, 5.2556E-002, 3.3190E-002,-9.9226E-002,
-3.4540E-002, 3.1598E-001, 5.3500E-001, 3.1598E-001,-3.4540E-002,
-9.9226E-002, 3.3190E-002, 5.2556E-002,-3.1035E-002,-3.0642E-002,
 2.8205E-002, 1.7465E-002,-2.4870E-002,-8.7185E-003, 2.1222E-002,
 2.7529E-003,-1.7460E-002, 1.2276E-003, 1.3778E-002,-3.6979E-003,
-1.0343E-002, 4.9986E-003, 7.2940E-003,-5.4097E-003,-4.7252E-003,
 5.1775E-003, 2.6882E-003,-4.5219E-003,-1.1901E-003,
 3.6353E-003 };
```

•

Graphic Equalizer Application Using TI's Linear Assembly Based FFT and IFFT

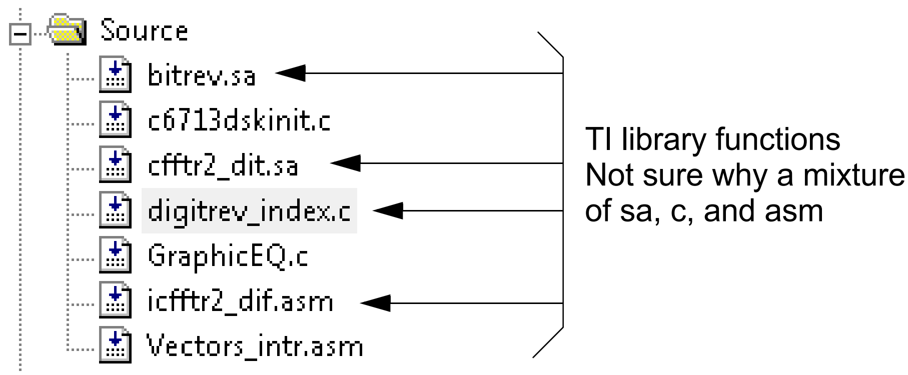
- This example is Chassaing's `GraphicEQ.c` program (Example 6.13, pp. 312ff)
- This program uses just an `iobuffer[PTS/2]` and a single working buffer, byte aligned working buffer `samples[PTS]`
- The fft twiddle factors are also held in a byte aligned array `W[PTS/RADIX]`

```
//GraphicEQ.c Graphic Equalizer using TI floating-point
//FFT functions

#include "DSK6713_AIC23.h"       //codec-DSK support file
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;//set sampling rate
#include <math.h>
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource=DSK6713_AIC23_INPUT_LINE; // select mic in
#include "GraphicEQcoeff.h"       //time-domain FIR coefficients
#define PI 3.14159265358979
```

```
#define PTS 256                   //number of points for FFT
//#define SQRT_PTS 16
#define RADIX 2
#define DELTA (2*PI)/PTS
typedef struct Complex_tag {float real,imag;} COMPLEX;
#pragma DATA_ALIGN(W,sizeof(COMPLEX))
#pragma DATA_ALIGN(samples,sizeof(COMPLEX))
#pragma DATA_ALIGN(h,sizeof(COMPLEX))
COMPLEX W[PTS/RADIX] ;                 //twiddle array
COMPLEX samples[PTS];
COMPLEX h[PTS];
COMPLEX bass[PTS], mid[PTS], treble[PTS];
short buffercount = 0;            //buffer count for iobuffer sam-
ples
float iobuffer[PTS/2];            //primary input/output buffer
float overlap[PTS/2];                //intermediate result buffer
short i;                          //index variable
short flag = 0;                   //set to indicate iobuffer full
float a, b;                       //variables for complex multiply
short NUMCOEFFS = sizeof(lpcoeff)/sizeof(float);
short iTwid[PTS/2] ;
float bass_gain = 1.0;            //initial gain values
float mid_gain = 0.0;             //change with GraphicEQ.gel
float treble_gain = 1.0;

interrupt void c_int11(void)      //ISR
{
 output_left_sample((short)(iobuffer[buffercount]));
 iobuffer[buffercount++] = (float)((short)input_left_sample());
 if (buffercount >= PTS/2)       //for overlap-add method iobuffer
  {                               //is half size of FFT used
   buffercount = 0;
   flag = 1;
  }
}

main()
{
 digitrev_index(iTwid, PTS/RADIX, RADIX);
 for( i = 0; i < PTS/RADIX; i++ )
```

```
  {
   W[i].real = cos(DELTA*i);
   W[i].imag = sin(DELTA*i);
  }
 bitrev(W, iTwid, PTS/RADIX);      //bit reverse W

 for (i=0 ; i<PTS ; i++)
  {
   bass[i].real = 0.0;
   bass[i].imag = 0.0;
   mid[i].real = 0.0;
   mid[i].imag = 0.0;
   treble[i].real = 0.0;
   treble[i].imag = 0.0;
  }
 for (i=0; i<NUMCOEFFS; i++)       //same # of coeff for each filter
  {
   bass[i].real = lpcoeff[i];      //lowpass coeff
   mid[i].real =  bpcoeff[i];      //bandpass coeff
   treble[i].real = hpcoeff[i];    //highpass coef
  }

 cfftr2_dit(bass,W,PTS);           //transform each band
 cfftr2_dit(mid,W,PTS);             //into frequency domain
 cfftr2_dit(treble,W,PTS);

 comm_intr();                      //initialise DSK, codec, McBSP
 while(1)                             //frame processing infinite
loop
  {
   while (flag == 0);              //wait for iobuffer full
          flag = 0;
   for (i=0 ; i<PTS/2 ; i++)       //iobuffer into samples buffer
    {
     samples[i].real = iobuffer[i];
     iobuffer[i] = overlap[i];     //previously processed output
    }                                 //to iobuffer
   for (i=0 ; i<PTS/2 ; i++)
    {                              //upper-half samples to overlap
     overlap[i] = samples[i+PTS/2].real;
```

```
     samples[i+PTS/2].real = 0.0; //zero-pad input from iobuffer
    }
   for (i=0 ; i<PTS ; i++)
     samples[i].imag = 0.0;       //init samples buffer

   cfftr2_dit(samples,W,PTS);

   for (i=0 ; i<PTS ; i++)        //construct freq domain filter
    {                             //sum of bass,mid,treble coeffs
    h[i].real = bass[i].real*bass_gain + mid[i].real*mid_gain
                                   + treble[i].real*treble_gain;
    h[i].imag = bass[i].imag*bass_gain + mid[i].imag*mid_gain
                                   + treble[i].imag*treble_gain;
    }
   for (i=0; i<PTS; i++)          //frequency-domain representation
    {                             //complex multiply samples by h
     a = samples[i].real;
     b = samples[i].imag;
     samples[i].real = h[i].real*a - h[i].imag*b;
     samples[i].imag = h[i].real*b + h[i].imag*a;
    }

   icfftr2_dif(samples,W,PTS);

   for (i=0 ; i<PTS ; i++)
      samples[i].real /= PTS;
   for (i=0 ; i<PTS/2 ; i++)      //add 1st half to overlap
      overlap[i] += samples[i].real;
  }                               //end of infinite loop
}                                 //end of main()
```



- This program is considerably more complicated than the C based FFT
- The basic signal processing functionality is an overlap and save transform domain filter algorithm with the FFT size $N = 128$
- The I/O buffer `iobuffer[128/2]` holds half the FFT length in samples

```
output_left_sample((short)(iobuffer[buffercount]));
iobuffer[buffercount++] = (float)((short)input_left_sample());
```

- Output D/A and A/D signal samples are exchanged in the interrupt routine in a common buffer
- The FFT and inverse FFT (IFFT) routines are taken from the TI linear assembly library: `cfftr2_dit.sa`, `icfftr2_dif.sa`, `bitrev.sa`, and `digitrev_index.c`
- Lowpass, bandpass, and highpass FIR filter coefficients are supplied from the file `GraphicEQ.h` and ultimately stored in the `COMPLEX` arrays 128 point arrays `bass[]`, `mid[]`, `treble[]` with zero padding (the actual filter lengths must not exceed 128)
- The three sets of FIR coefficients are transformed (once) into

the frequency domain, and then combined via weighting into a composite transform domain filtering function

```
for (i=0; i<NUMCOEFFS; i++)      //same # of coeff for each filter
  {
   bass[i].real = lpcoeff[i];     //lowpass coeff
   mid[i].real =  bpcoeff[i];     //bandpass coeff
   treble[i].real = hpcoeff[i];   //highpass coef
  }

 cfftr2_dit(bass,W,PTS);          //transform each band
 cfftr2_dit(mid,W,PTS);            //into frequency domain
 cfftr2_dit(treble,W,PTS);
```

- The input signal samples are transferred to the `COMPLEX` processing buffer `samples[128]` with the upper half of this array containing only zeros before the FFT `cfftr2_dit()`
- The actual transform domain filtering is performed with the code

```
 for (i=0; i<PTS; i++)          //frequency-domain representation
   {                            //complex multiply samples by h
    a = samples[i].real;
    b = samples[i].imag;
    samples[i].real = h[i].real*a - h[i].imag*b;
    samples[i].imag = h[i].real*b + h[i].imag*a;
   }
  // Finally inverse transform one block
  icfftr2_dif(samples,W,PTS);
```

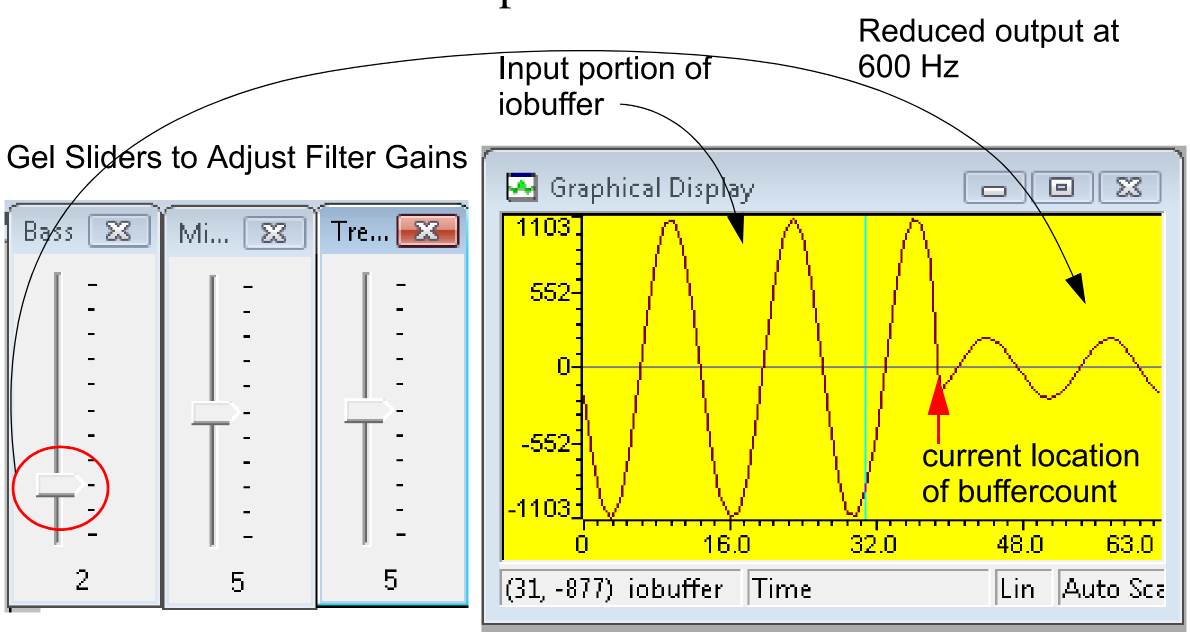
- The forward transformed data is returned in bit reverse order by `cfftr2_dit()`, but the routine `cicfftr2_dif()` expects a bit reversed input and returns normal ordered data, so all is well with this mixture of FFT/IFFT routines
- The only remaining detail is that each frame of I/O data con-

tains only 64 points, so the extra 64 points, that form the overlap, must be saved and added to the next frame of output data

```
//GraphicEQcoeff.h Filter coefficients for Graphic Equalizer
//obtained with Matlab fir1()

float lpcoeff[] = { 7.8090987e-004,  5.5772256e-004, -2.9099535e-004,
-1.0680710e-003, -8.9417753e-004,  4.4134422e-004,  1.8660902e-003,
1.6586856e-003, -7.0756083e-004, -3.2950637e-003, -2.9970618e-003,
1.0674784e-003,  5.5037549e-003,  5.1029563e-003, -1.4905208e-003,
-8.7103609e-003, -8.2694434e-003,  1.9403706e-003,  1.3299499e-002,
1.3018139e-002, -2.3781611e-003, -2.0079912e-002, -2.0467002e-002,
2.7659079e-003,  3.1120688e-002,  3.3649522e-002, -3.0698723e-003,
-5.3589440e-002, -6.4772988e-002,  3.2635553e-003,  1.3809973e-001,
2.7310671e-001,  3.2967515e-001,  2.7310671e-001,  1.3809973e-001,
3.2635553e-003, -6.4772988e-002, -5.3589440e-002, -3.0698723e-003,
3.3649522e-002,  3.1120688e-002,  2.7659079e-003, -2.0467002e-002,
-2.0079912e-002, -2.3781611e-003,  1.3018139e-002,  1.3299499e-002,
1.9403706e-003, -8.2694434e-003, -8.7103609e-003, -1.4905208e-003,
5.1029563e-003,  5.5037549e-003,  1.0674784e-003, -2.9970618e-003,
-3.2950637e-003, -7.0756083e-004,  1.6586856e-003,  1.8660902e-003,
4.4134422e-004, -8.9417753e-004, -1.0680710e-003, -2.9099535e-004,
5.5772256e-004,  7.8090987e-004 };

float bpcoeff[] = { -1.0740274e-003,  2.7910515e-004, -2.6262359e-004,
6.0234498e-004,  2.1987775e-003, -1.2895387e-003, -2.7954474e-003,
6.8734556e-004, -6.6338736e-004,  1.4571032e-003,  7.1033754e-003,
-3.1523737e-003, -8.9090924e-003,  1.6470428e-003, -1.4383367e-003,
2.8105100e-003,  1.8819817e-002, -5.7758163e-003, -2.3101173e-002,
2.9409437e-003, -2.3416627e-003,  4.1322990e-003,  4.4546556e-002,
-8.2767312e-003, -5.7646558e-002,  4.1799488e-003, -3.0590719e-003,
4.9333674e-003,  1.3424490e-001, -9.7916543e-003, -2.7127645e-001,
4.9406664e-003,  3.2981693e-001,  4.9406664e-003, -2.7127645e-001,
-9.7916543e-003,  1.3424490e-001,  4.9333674e-003, -3.0590719e-003,
4.1799488e-003, -5.7646558e-002, -8.2767312e-003,  4.4546556e-002,
4.1322990e-003, -2.3416627e-003,  2.9409437e-003, -2.3101173e-002,
-5.7758163e-003,  1.8819817e-002,  2.8105100e-003, -1.4383367e-003,
1.6470428e-003, -8.9090924e-003, -3.1523737e-003,  7.1033754e-003,
1.4571032e-003, -6.6338736e-004,  6.8734556e-004, -2.7954474e-003,
-1.2895387e-003,  2.1987775e-003,  6.0234498e-004, -2.6262359e-004,
2.7910515e-004, -1.0740274e-003 };

float hpcoeff[] = { 2.9251723e-004, -8.3631146e-004,  5.5324390e-004,
4.6576424e-004, -1.3030373e-003,  8.4723869e-004,  9.2771594e-004,
```

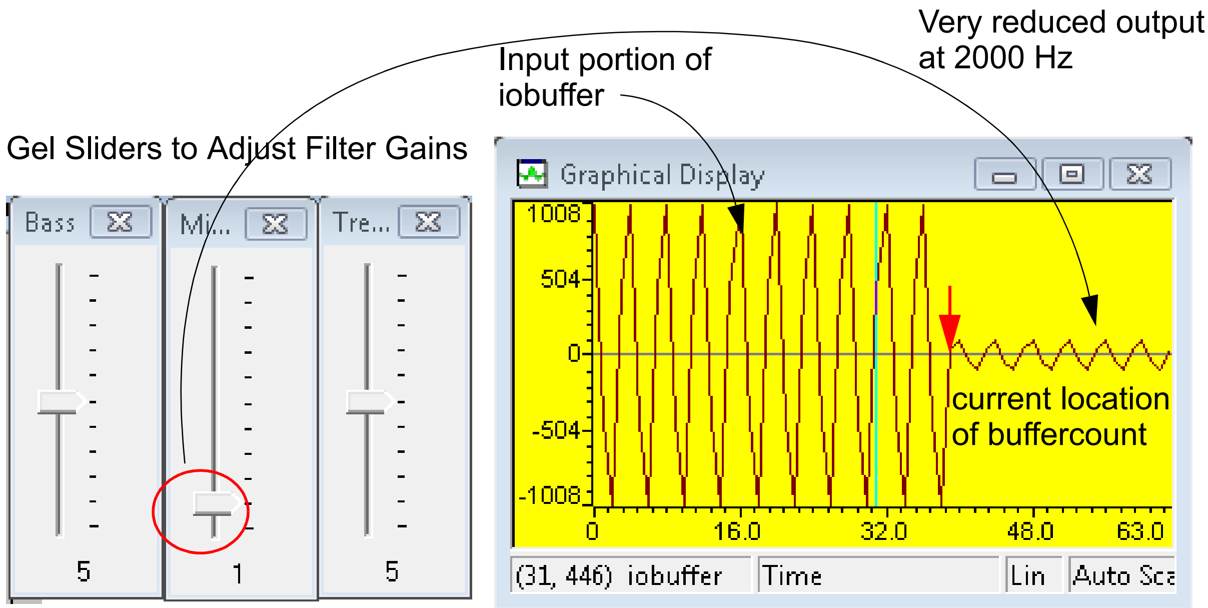
```
-2.3446248e-003,  1.3700139e-003,  1.8377162e-003, -4.1013166e-003,
2.0825534e-003,  3.3998966e-003, -6.7460946e-003,  2.9268523e-003,
5.8982643e-003, -1.0537290e-002,  3.8311475e-003,  9.7871064e-003,
-1.5950261e-002,  4.7165823e-003,  1.5941836e-002, -2.4049009e-002,
5.5046570e-003,  2.6488538e-002, -3.7809758e-002,  6.1247271e-003,
4.8635148e-002, -6.9381330e-002,  6.5208000e-003,  1.3299709e-001,
-2.7791356e-001,  3.3950443e-001, -2.7791356e-001,  1.3299709e-001,
6.5208000e-003, -6.9381330e-002,  4.8635148e-002,  6.1247271e-003,
-3.7809758e-002,  2.6488538e-002,  5.5046570e-003, -2.4049009e-002,
1.5941836e-002,  4.7165823e-003, -1.5950261e-002,  9.7871064e-003,
3.8311475e-003, -1.0537290e-002,  5.8982643e-003,  2.9268523e-003,
-6.7460946e-003,  3.3998966e-003,  2.0825534e-003, -4.1013166e-003,
1.8377162e-003,  1.3700139e-003, -2.3446248e-003,  9.2771594e-004,
8.4723869e-004, -1.3030373e-003,  4.6576424e-004,  5.5324390e-004,
-8.3631146e-004,  2.9251723e-004 };
```

- Observe `iobuffer` for a 600 Hz input tone with the bass slider set below the midpoint



Random CCS stop and see a mixture of input and output samples in iobuffer for a 600 Hz input.

- Repeat with a 2000 Hz input and the mid slider above the midrange point



Random CCS stop and see a mixture of input and output samples in iobuffer for a 2000 Hz input.

Using Direct Memory Access (DMA)[1]

- The most efficient way to implement frame-based processing for real-time DSP is to take advantage of the processors DMA architecture
  - With DMA we can transfer data from one memory location to another without any work being required by the CPU
  - Data can be transferred one location at a time or in terms of blocks

1. T. Welch, C. Wright, and M. Morrow, Real-Time Digital Signal Processing, CRC Press, Boca Raton, FL, 2006. ISBN 0-8493-7382-4

- The Welch text contains a nice example of doing exactly this for the C6713 DSK
- The DMA hardware needs to first be configured with source and destination memory locations and the number of transfers to perform
- The triple buffering scheme can be implemented in DMA
- On the C6713 the DMA is termed EDMA, which as stated in the hardware overview of Chapter 2, the ‘E’ stands for *enhanced*
- When using DMA an important factor is keeping the memory transfer synchronized with processor interrupts which are firing relative to the sampling rate clock
- The EDMA can be configured to behave similar to the CPU interrupts, thus allowing the buffers to remain synchronized
- A note on using the Welch programs:
  - The AIC23 codec library in their programs is different from the Chassaing text
  - Their code is more structured than the Chassaing text, so once adopted, it is easier to work with
  - The EDMA example is geared toward general frames based processing, but presumably can be modified to work with the TI FFT library