EL6183: LAB No. 7

NYU Poly; Spring 2011

Discrete Time Fourier Transform (DTFT), Discrete Fourier Transform (DFT), and Fast Fourier Transform (FFT) Implementations.

Ayan Shafqat (0309862)
4/11/2011

Introduction

The Difference between Discrete Time Fourier Transform (DTFT), Discrete Fourier Transform (DFT), and Fast Fourier Transform (FFT)

Mémoiresur la propagation de la chaleurdans les corps solides was one of the revolutionary ideas that were published in the early 19th century by Joseph Fourier. This article stated that, any function, as long as they are periodic, can be expressed as summed series of trigonometric functions. Today this method of transform is known as "Classical Fourier Series Analysis," which is used very often in signal analysis. Although there were methods to perform harmonic analysis at that time, Fourier described harmonic analysis in a very elegant formula, and later has led to the development of Fourier Transform.

Fourier Transform takes a function that is in time domain and basically calculates how much of a certain frequency is present in that function. When we plot the magnitude of the function in frequency domain, we actually see a density function that is calculated by using Fourier Transform. In a continuous time signal, Fourier Transform is defined as:

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt$$

As observed in the formula above, this would be very impractical to be used in signal processing, since in order for $f(t)$ to be truly continuous, $f(t)$ must be deterministic. Exact mathematical formula has to be known that truly describes $f(t)$, which is never possible in real-time applications. Therefore, samples or chunks of points at equal time interval is taken, in order the function to be approximately represented in discrete time. Thus, Fourier Transform becomes Discrete Time Fourier Transform or DTFT, which is known as:

$$F(\omega) = \sum_{n=-\infty}^{\infty} f[n]e^{-j\omega n}$$

This poses a problem in terms of signal processing, since ω is still continuous. In addition, this transform requires an infinite number of discrete terms in order to calculate DTFT, which is not practical for signal processing, or anything for that matter.

As observed in the formula for DTFT, $e^{j\omega n}$ is an unit circle and as n goes from $\pm\infty$, $F(\omega)$ also spins around the unit circle. Therefore, $F(\omega)$ is periodic with a period of 2π. Taking this property into account, it can be stated that anything beyond 2π can be ignored and the unit circle can be sampled at N points. Thus, $F(\omega)$ becomes $F(k)$ in which $\omega = 2\pi/N$, giving birth to the idea of Discrete Fourier Transform, or DFT. DFT is expressed as:

$$F(k) = \sum_{n=0}^{N-1} f[n]e^{-j\left(\frac{2\pi k}{N}\right)n} \text{ for } k = 0, 1, 2, \dots, N-1$$

From the formula above, it can be noted this is a very elaborative process that takes $(N-1)^2$ complex multiplication and $N(N-1)$ complex addition. This can take a long time in a slow CPU, for which there are algorithms available for Fast Fourier Transform or FFT.

FFT is not an approximation to DFT; rather FFT gives the exact result as DFT. But, FFT uses certain properties of DFT to calculate it much quicker. There are many FFT algorithms available, such as the Cooley-Tukey Algorithm, Radix-2 Algorithm, etc.

Although these FFT algorithms are very fast, FFT algorithms require all the data to be present in the buffer before the algorithm can proceed. Therefore, I have made some modifications to DFT formula in order to be calculated as the CPU is receiving samples, which I called "Real Time DFT by Utilization of Processor Time during Interrupt."

Real Time DFT by Utilization of Processor Time during Interrupt

First of all, let me state that this algorithm only works for real signals only, by using event model of an interrupt. According to the formula for DFT:

$$F(k) = \sum_{n=0}^{N-1} f[n]e^{-j\left(\frac{2\pi k}{N}\right)n} \text{ for } k = 0, 1, 2, \dots, N-1$$

While $f[n]$ is being received one at a time. Therefore, we can call $f[n]$ an event, which occur every $1/f_s$ second. Also, we can observe from the formula above that:

$$F(k) = \begin{bmatrix} f[0] \\ \vdots \\ f[N-1] \end{bmatrix} \begin{bmatrix} 1 & \dots & (N-1) \\ \vdots & \ddots & \vdots \\ 1 & \cdots & e^{-j(2\pi N)} \end{bmatrix}$$

$$F(0) = f[0]e^{-j0} + f[1]e^{-j0} + \cdots\cdots + f[N-1]e^{-j0}$$

By observing the matrix above, we can see that we can add the k elements until (N – 1) every time we receive a sample. Therefore, the algorithm becomes:

$$\begin{array}{ll} F[k] = 0 & \text{for } k = 0 \; to \; N-1 \\ s \leftarrow input\_sample(\;\;) & \\ F[0] \leftarrow F[0] + s & k = 0 \;\therefore W_N^{-k} = 1 \\ F[k] \leftarrow F[k] + s * e^{-j\left(\frac{2\pi k}{N}\right)n} & \text{for } k = 1 \; to (N-1) \\ n \leftarrow n + 1 & Wait\ for\ next\ sample \end{array}$$

Since we know that given a real signal$f[n]$, the real components of F[k] are symmetric and the imaginary components are anti-symmetric; weonly then perform half of the calculations:

$$\begin{array}{ll} F[k] \leftarrow F[k] + s * e^{-j\left(\frac{2\pi k}{N}\right)n} & \text{for } k = 0 \; to \left(\frac{N}{2} - 1\right) \\ n \leftarrow n + 1 & Wait\ for\ next\ sample \\ When\ n = (N-1) & \\ F[N-1-k] \leftarrow \text{conj}(F[k]) & \text{for } k = 0 \; to \left(\frac{N}{2} - 1\right) \end{array}$$

Implementation

This algorithm was implemented in C and was tested on both Intel (Intel Atom N270) and TMS320C6711 DSK platform. The code is compared to the FFT output of MATLAB to a similar time domain data. There were couple of issues regarding the accuracy of the result, but later it was discovered that it was due to precision used in order to calculate DFT. The C code implemented used single precision data type (C float) while MATLAB uses double precision data type (C double). Obviously double precision will correspond to more accurate result, but it can also add a lot to the processor time. Therefore, the topic about precision is ignored, since real-time is the main priority.

Implementation (C-Header)

```c
/*******************************************************************************
 * Real-Time DFT by Utilizing Processor Time Between Interrupt                 *
 *******************************************************************************
 * This is a simple algorithm for calculating DFT while the CPU waits for the  *
 * next sample. By doing so, a N-point DFT can be resulted by the time the     *
 * processor has received all N-point signals.                                 *
 *                                                                             *
 * Please note: This is not Goertzel algorithm, but a simple optimization of   *
 * DFT algorithm presented in EL6183 DSP Lab #7 manual.                        *
 *                                                                             *
 * Author: Ayan Shafqat (0309862)                                              *
 * Polytechnic Institute of NYU, Spring 2011                                   *
 ******************************************************************************/
#ifndef RTDFT_H
#define RTDFT_H
/* Necessary Macros */
#include <math.h>                // Needed for sin(x) and cos(x)
#define DFT_PRECISION double     // Use the proper precision for calculations
#define c_real( x )    x[0]      // Pull real data from dft_complex
#define c_imag( x )    x[1]      // Pull imag data from dft_complex
#define DFT_INCOMPLETE 0x00      // DFT FLAG
#define DFT_COMPLETE   0x01      // DFT FLAG
#define DFT_ERROR      0x02      // DFT FLAG
#define DFT_TWOPI      6.283185307179586
#define DFT_LIN_MAGNITUDE 0x00  // FLAG
#define DFT_LOG_MAGNITUDE 0x01  // FLAG
#define dft_c_abs( x ) sqrt(c_real( x )*c_real( x ) + c_imag( x )*c_imag( x ))
/* Custom data types */
typedef DFT_PRECISION dft_complex[2]; // Complex data type for DFT
typedef struct {
    unsigned int  N_dft;         // Size of DFT
    unsigned char DFT_flag;      // DFT complete flag
    dft_complex *dft_output;     // Pointer to DFT output (complex)
} dft_plan;
typedef char dft_flag;           // Flag data type
unsigned int __DFT_COUNTER = 0; // DFT COUNTER: DO NOT REMOVE
/******************************************************************************/
/* Functions Declarations                                                     */
/******************************************************************************/
dft_flag create_dft_plan(dft_plan*,dft_complex*,unsigned int);
dft_flag rdft_calc_interrupt(dft_plan*,DFT_PRECISION);
dft_flag recalculate_dft(dft_plan*);
void DFTShiftCalcMagPhaseFreq(dft_complex*,DFT_PRECISION*,DFT_PRECISION*,
                        DFT_PRECISION*,unsigned int,unsigned int,unsigned char);
/******************************************************************************/
/* Function Implementation                                                    */
/******************************************************************************/
dft_flag create_dft_plan(dft_plan* dft, dft_complex* output, unsigned int dft_size) {
    /* create_dft_plan returns a pointer to DFT_KERNEL Struct, which can
     * then be used by rdft_calc_interrupt(kernel*) to calculate DFT
     * inside the interrupt */
    register unsigned int n, k;
    if(output == NULL) return DFT_ERROR; // Error: Output array isn't allocated
    dft->N_dft       = dft_size;
    dft->DFT_flag    = DFT_INCOMPLETE;
    dft->dft_output  = output;
    for(k = 0; k < dft_size; k++) {
        c_real(dft->dft_output[k]) = 0.0;
        c_imag(dft->dft_output[k]) = 0.0;}
    return (0); // No error: For now
}
```

```
dft_flag rdft_calc_interrupt(dft_plan* dft, DFT_PRECISION input) {
    /* The key to use this function is to place it inside the interrupt. This
     * function calculates DFT as it is receiving the samples. After the desired
     * length is received, this function stops calculating and returns 1. Other
     * than that, this function returns zero (no error).*/
    DFT_PRECISION digital_frequency;
    register unsigned int N, k;
    N = dft->N_dft;
    if(dft == NULL) return DFT_ERROR; // Error: No kernel defined
    if(N%2 != 0   ) return DFT_ERROR; // Not a multiple of 2 Error
    if(__DFT_COUNTER < (dft->N_dft) && (dft->DFT_flag) != DFT_COMPLETE) {
        c_real(dft->dft_output[0]) += input; // k = 0; exp(-j2piKn/N) = 1
        for(k = 1; k < N/2+1; k++) {
            digital_frequency = -DFT_TWOPI * (DFT_PRECISION)k *
                                (DFT_PRECISION)__DFT_COUNTER / (DFT_PRECISION)(N);
            c_real(dft->dft_output[k]) += (input) * cos(digital_frequency);
            c_imag(dft->dft_output[k]) += (input) * sin(digital_frequency);
        }
        __DFT_COUNTER++;
    } else {
        dft->DFT_flag = DFT_COMPLETE;
        __DFT_COUNTER = 0;
        return 1; /* DFT FINISHED */
    }
    if(__DFT_COUNTER == (N - 1)) {
   /* By using one of DFT Properties, we can eliminate half of the calculations
    * PROPERTY: Given x[n] is real, X_r[k] -- Symmetric, and X_i[k] -- Anti-symmetric.*/
        for(k = 0; k < N/2 + 1; k++) {
            c_real(dft->dft_output[N - 1 - k]) =  c_real(dft->dft_output[k]);
            c_imag(dft->dft_output[N - 1 - k]) = -c_imag(dft->dft_output[k]);
        }
    }
    return 0;
}
dft_flag recalculate(dft_plan* dft) {
    /* This function is to be used when dft needs to be calculated again, such
     * as a real time spectrum analyzer. This can be placed in the control loop
     * which controls the interrupt. (1) Resets __DFT_COUNTER to zero, (2) and
     * zeros out the dft values. (3) Lastly changes the flag*/
    int k;
    __DFT_COUNTER = 0;  // Resetting DFT_COUNTER
    for(k = 0; k < dft->N_dft; k++) {
        c_real(dft->dft_output[k]) = 0.0; // Reset DFT vector
        c_imag(dft->dft_output[k]) = 0.0; //   "    "    "
    }
    dft->DFT_flag = DFT_INCOMPLETE;   // Resetting dft plan's flag
    return 0;
}
void DFTShiftCalcMagPhaseFreq(dft_complex* dftdat, DFT_PRECISION* dft_magn,
DFT_PRECISION* dft_angle, DFT_PRECISION* frequency, unsigned int dft_length,
unsigned int fs, unsigned char flag) {
    /* This function calculates magnitude and phase of DFT and also uses proper
     * shifting such that zero is the center frequency. This function also
     * calculates the frequency axis as well. (Not recommended for real-time
     * use). Only for viewing purpose. */
    register unsigned int n, half_dft_length;
    half_dft_length = dft_length/2;
    dft_complex temp_dft;
```

C-Header: RTDFT.H (Continued)

```
    /* Reordering for human viewing */
    for(n = 0; n < half_dft_length; n++) {
        /* Reordering magnitude 0 is the center frequency*/
        c_real(temp_dft) = c_real(dftdat[n]);
        c_imag(temp_dft) = c_imag(dftdat[n]);
        c_real(dftdat[n]) = c_real(dftdat[half_dft_length + n]);
        c_imag(dftdat[n]) = c_imag(dftdat[half_dft_length + n]);
        c_real(dftdat[half_dft_length + n]) = c_real(temp_dft);
        c_imag(dftdat[half_dft_length + n]) = c_imag(temp_dft);
    }
    for(n = 0; n < dft_length; n++){
        // Calculating Frequency axis
        frequency[n] = (float)(n*fs/dft_length) - (float)fs/2.0;
        // Calculating Phase
        dft_angle[n] = (atan2(c_imag(dftdat[n]),c_real(dftdat[n])));
        // Calculating magnitude according to flag (LINEAR or LOG)
        if(flag == DFT_LOG_MAGNITUDE)
            dft_magn[n] = 10* log10(dft_c_abs(dftdat[n]));
        else
            dft_magn[n] = dft_c_abs(dftdat[n]);
    }
}
#endif /* RFT_H */
```

Benchmarks (Using PC)

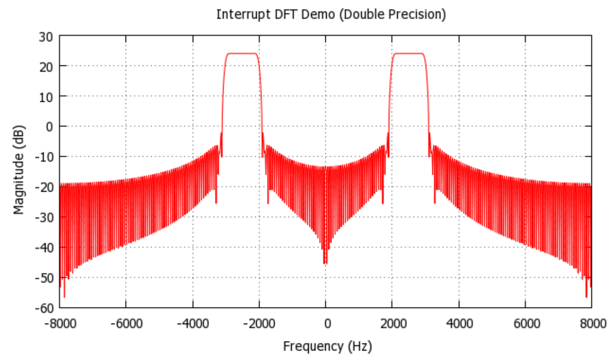
```
+-----------------------------------------+
| Benchmark Testing                       |
| Interrupt Based DFT Calculation         |
| for embedded platforms                  |
| CPU SPEED:  ~    18.77565MFLOP/S        |
+-----------------------------------------+
| Performing DFT of size    256           |
| Process took: ~         0.28 MFlOps     |
+-----------------------------------------+
| Performing DFT of size    512           |
| Process took: ~         1.76 MFlOps     |
+-----------------------------------------+
| Performing DFT of size   1024           |
| Process took: ~         7.92 MFlOps     |
+-----------------------------------------+
| Performing DFT of size   2048           |
| Process took: ~        31.97 MFlOps     |
+-----------------------------------------+
| Performing DFT of size   4096           |
| Process took: ~       125.85 MFlOps     |
+-----------------------------------------+
| Performing DFT of size   8192           |
| Process took: ~       504.60 MFlOps     |
+-----------------------------------------+
```

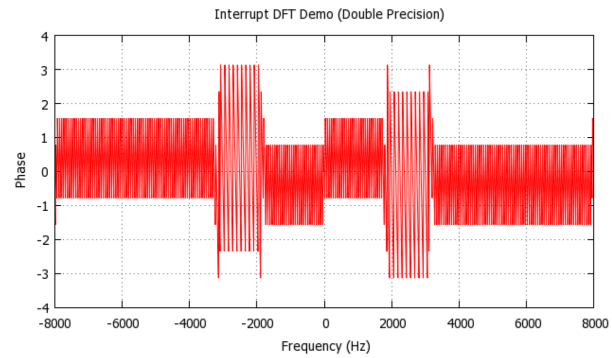
```
+-----------------------------------------+
| Benchmark Testing                       |
| Interrupt Based DFT Calculation         |
| for embedded platforms                  |
| CPU SPEED:  ~    50.51620MFLOP/S        |
+-----------------------------------------+
| Performing DFT of size    256           |
| Process took: ~         0.81 MFlOps     |
+-----------------------------------------+
| Performing DFT of size    512           |
| Process took: ~         1.57 MFlOps     |
+-----------------------------------------+
| Performing DFT of size   1024           |
| Process took: ~         4.75 MFlOps     |
+-----------------------------------------+
| Performing DFT of size   2048           |
| Process took: ~        21.27 MFlOps     |
+-----------------------------------------+
| Performing DFT of size   4096           |
| Process took: ~        84.31 MFlOps     |
+-----------------------------------------+
| Performing DFT of size   8192           |
| Process took: ~       335.73 MFlOps     |
+-----------------------------------------+
```

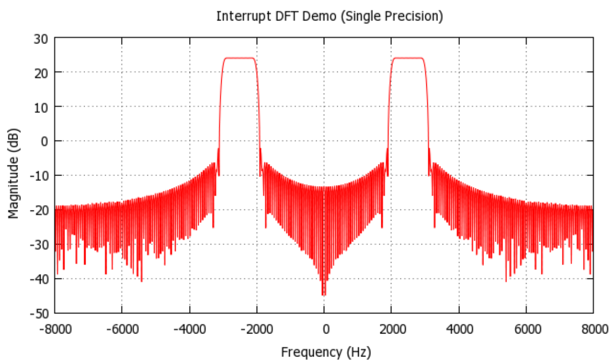
Observations:

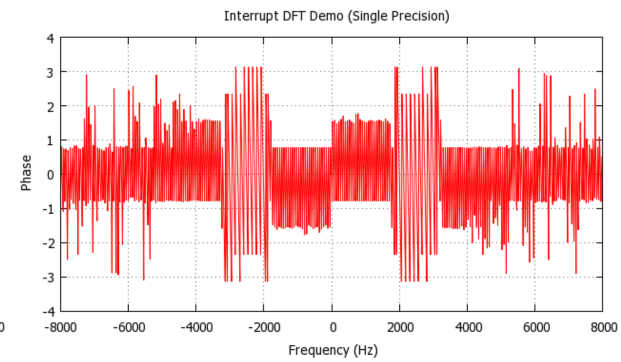
According to the benchmarks above, this algorithm is about 97% faster than calculating DFT via brute force. But, the benchmark tools in `<time.h>`were not sufficient enough to confirm this is true, since it lacked accuracy. For some benchmark tests, the outputs were zero, which is not possible. But, a more realistic test result can be made using the DSK board. Again, precision was an issue, but it can be ignored.

Outputs (C Code)

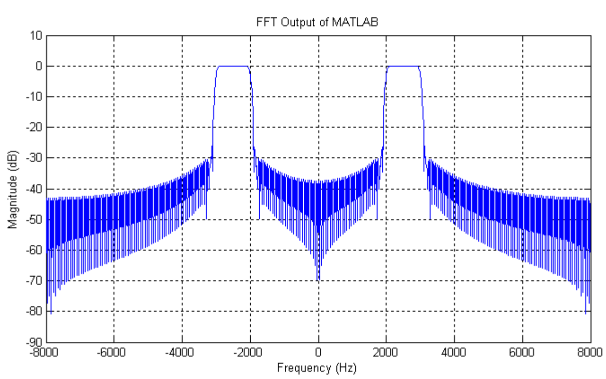


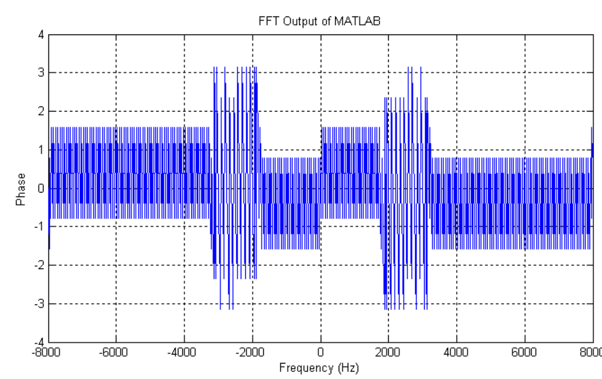




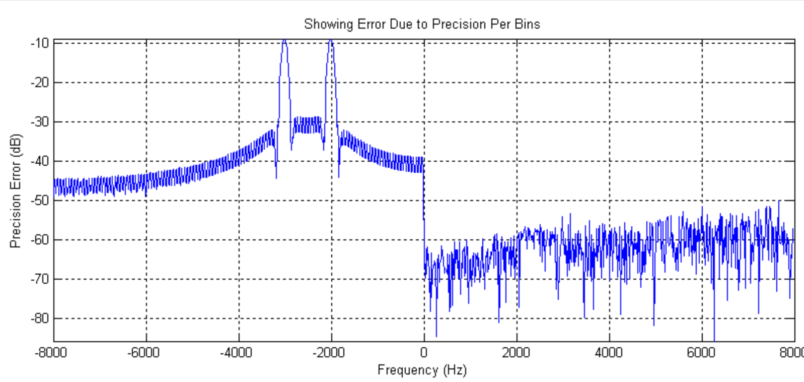


Comparing to FFT of MATLAB





Error Analysis



Appendix A (Benchmark Test Code written for PC)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <windows.h>
#include "RTDFT.h"
#define SET_PLOT_PRECISION FLOAT32
#include "gnuplotpipe.h"
#include "BP_FILTER.h"
#define PLOT_DFT_DATA
void pause(void);
double randfloat(void);
double get_FLOPS(void);
int main(void) {
    /* Real Time Discrete Fourier Transform Test program */
    /* RTDFT variables: */
    dft_plan DFT;
    unsigned int i, n, N;
    dft_complex *out,*dummy;
    DFT_PRECISION input;
#ifdef BENCHMARK_RDFT
    /* Benchmark variables */
    DWORD c_start, c_end;
    double cpu_flops, flop;
    cpu_flops = get_FLOPS();  printf("\n\n");
    printf("+--------------------------------+\n");
    printf("| Benchmark Testing              |\n");
    printf("| Interrupt Based DFT Calculation|\n");
    printf("| for embedded platforms         |\n");
    printf("| CPU SPEED:  ~% 11.5fMFLOP/S|\n", (cpu_flops/1e+6));
    printf("+--------------------------------+\n");
    for(n = 256; n < (1<<14); n *= 2){
        /* BEGIN BENCHMARK */
        dummy = malloc(n*sizeof(dft_complex));
        printf("| Performing DFT of size % 5d   |\n", n);
        c_start = GetTickCount();
        create_dft_plan(&DFT, dummy, n); // Sort out and allocate
        for(i = 0; i < n; i++) {
            if(i < FIR_FILTER_LENGTH) input = h_bp_filter[i];
            else input = 0;
            rdft_calc_interrupt(&DFT, input);}
        c_end = GetTickCount();
        flop = ((double)c_end - (double)c_start)*cpu_flops*1e-9;
        printf("| Process took: ~% 9.2f MFlOps|\n", flop);
        printf("+--------------------------------+\n");
        recalculate(&DFT);
        free(dummy);    }
    printf("\n\n");
    pause();
#endif
#ifdef PLOT_DFT_DATA
    N = 1024; out = malloc(N*sizeof(dft_complex));
    create_dft_plan(&DFT, out, N); // Sort out and allocate
    for(i = 0; i < N; i++) {
        if(i < FIR_FILTER_LENGTH) input = h_bp_filter[i]*(1<<8);
        else input = 0;
        rdft_calc_interrupt(&DFT, input);} // End of loop
    DFT_PRECISION *dft_magn, *dft_freq, *dft_angle;
    dft_magn = malloc(N*sizeof(DFT_PRECISION)); dft_freq = malloc(N*sizeof(DFT_PRECISION));
    dft_angle = malloc(N*sizeof(DFT_PRECISION));
    DFTShiftCalcMagPhaseFreq(out, dft_magn, dft_angle, dft_freq, N, 16000, DFT_LOG_MAGNITUDE);
    free(out); open_gnuplot_pipe();
    if(GNUPLOT_EXISTS) {
        printf("\n\n=[Plotting]=====================\n");
        printf("Plotting magnitude of DFT\n");
        gnuplot_xy(dft_freq, dft_magn, N);
        gnuplot_xy_label("Frequency (Hz)", "Magnitude (dB)");
        gnuplot_title("Interrupt DFT Demo (Single Precision)");
        pause();
        printf("Plotting Phase of DFT\n");
```

```
        gnuplot_xy(dft_freq, dft_angle, N);
        gnuplot_xy_label("Frequency (Hz)", "Phase");
        gnuplot_title("Interrupt DFT Demo (Single Precision)");
        pause();
        printf("=[Exit]========================\n\n");
        close_gnuplot_pipe();
    } free(dft_magn); free(dft_freq); free(dft_angle);
#endif
    return 0;
}
void pause(void)       { fprintf(stderr,"Press any key to continue.\n"); getch(); }
double randfloat(void) { return rand()/((double)(RAND_MAX)+1);}
double get_FLOPS(void) {
    register unsigned int N = (1<<24), n; volatile double R, avg = 0;
    volatile DWORD c_start, c_end;
    c_start = GetTickCount();
    for(n = 0; n < N; n++) R = randfloat() + randfloat();
    c_end   = GetTickCount();
    avg += (double)c_end - (double)c_start;
    c_start = GetTickCount();
    for(n = 0; n < N; n++) R = randfloat() - randfloat();
    c_end   = GetTickCount();
    avg += (double)c_end - (double)c_start;
    c_start = GetTickCount();
    for(n = 0; n < N; n++) R = randfloat() * randfloat();
    c_end   = GetTickCount();
    avg += (double)c_end - (double)c_start;
    c_start = GetTickCount();
    for(n = 0; n < N; n++) R = randfloat() / randfloat();
    c_end   = GetTickCount();
    avg += (double)c_end - (double)c_start;
    avg = (16*N)/(avg*1e-3);
    return avg;
}
```

Test Code Written for TMS3206711 DSK

```
#include "DSK6713_AIC23.h"                // Chip codec support
Uint32 fs=DSK6713_AIC23_FREQ_16KHZ;       // Set sampling rate (16 KHz)
#define DSK6713_AIC23_INPUT_MIC 0x0015    // Define input source
#define DSK6713_AIC23_INPUT_LINE 0x0011   // Define input source (LINE)
Uint16 inputsource=DSK6713_AIC23_INPUT_MIC; // Select input
#include "BP_FILTER.H"
#include "RTDFT.H"
// Global variables initialization
dft_kernel DFT;
unsigned int N = 1024;
dft_complex dft_out[N];
dft_flag process_finished;
unsigned short n = 0;
// These are the variables to watch
float dft_magn[N], dft_freq[N], dft_angle[N];

interrupt void c_int11() {
    if(n < FIR_FILTER_LENGTH) input = h_bp_filter[n];
    else input = 0; n++;      // Zero padding signal
    process_finished = rdft_calc_interrupt(&DFT, input);
}
void main(){
    create_dft_kernel(&DFT, dft_out, N);
    comm_intr();
    while(!process_finished);
    DFTShiftCalcMagnPhaseFreq(out, dft_magn, dft_angle, dft_freq, N, fs, DFT_LOG_MAGNITUDE);

}
```