**MEMORANDUM**

**To:** Dr. Thornburg

**From:** Ryan Wise and Paul Hertfelder

**Date:** 12/04/2017

**Subject: Lab 4 DSP**

**Introduction**

The purpose of this lab was to use the mips\_fft16() function that is built into the DSP library to do forward and inverse FFT calculations. The second objective of this lab was to perform FFT convolution by replacing the time-domain convolution algorithm with multiplication in the frequency domain.

**Design**

The code was written in MPLAB X and the analog discovery was used to create a signal that would be passed through an analog filter. Three filters were used one being an ideal low pass filter, the second being an ideal high pass filter and the third being an ideal all pass filter. The same reconstruction filter designed in Lab 2A is reused in this lab. (The reconstruction filter in Lab 2A had a cutoff frequency of around 7 kHz.)

The code was written mostly from scratch, only using the timer function provided in Lab 3 as a basis to build upon. The idea is to read in a signal on the ADC, scale this signal to Q0.15 format, use the mips FFT function, perform filtering through multiplication in the frequency domain, undo the fft using the same mips FFT function, scale the result, and output the waveform. All of this is implemented in MPLAB X. The first version takes 8 samples spread out by the timer function, pads 8 zeros to the end, and performs the FFT. The current method used is to take a single sample at each time interval and shift the original data to act as a continuous signal, followed by the same padding and FFT as used before.

The filters are built each time the code is uploaded, so, depending on the current test to be performed, the filter kernel will be different. The filter is 16 terms long and is multiplied by each term of the result of the forward FFT. Frequency spectrums of filters are centered about 0 frequency. The frequency components of a digitally converted signal repeat every sampling frequency, and start at negative half of the sampling frequency. That means that the negative portion of the signal that must also be passed is located in the second half of the output of the forward FFT. The filter arrays must therefore act as if mirrored over the 8th data point. For example, rather than a low pass filter with an array of [1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0], we use an array of [1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1].

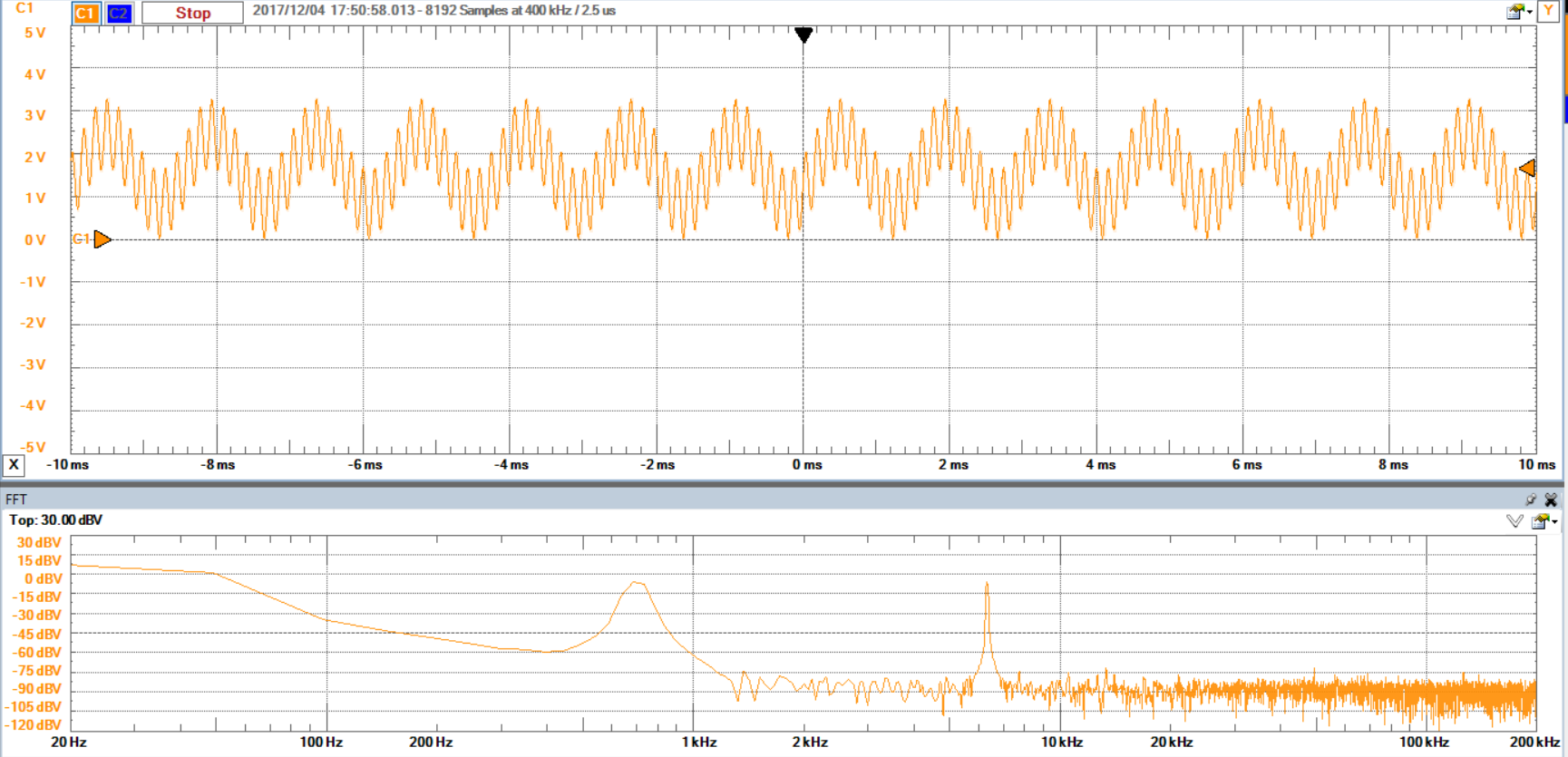
**Testing & Results**

In comparing the two methodologies mentioned in design, the single data point at a time appears to more closely match the original input signal. They both do a decent job, but the 8 samples at a time results in additional frequency components that affect the signal.

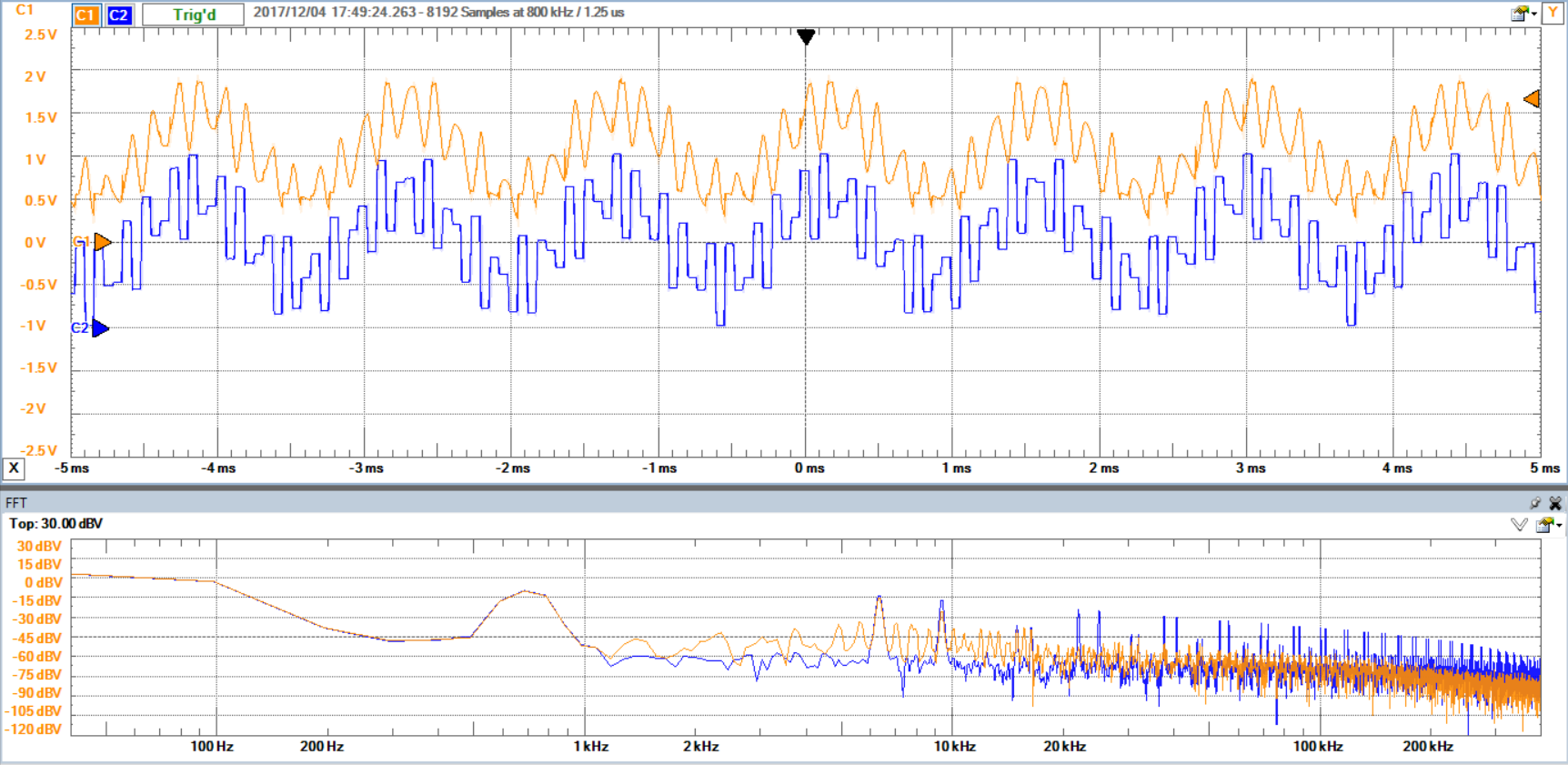
Filtering appears mostly unsuccessful no matter the changes made. Using a sample rate of 15.6 kHz, each frequency point stored after the forward FFT should represent 975 Hz of frequency data. This means that a filter of the form [ 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 ] should pass signals below 1.95 kHz, and remove any others. Inputting a signal with frequency components of 700 Hz and 6.2 kHz produces an output waveform with a strong 700 Hz signal and slightly smaller 6.2 kHz signal. The filter should have completely removed the 6.2 kHz portion. The high pass filter has little discernible frequency components, just large components at times. We have remained unable to solve the filtering issue.

**Conclusion**

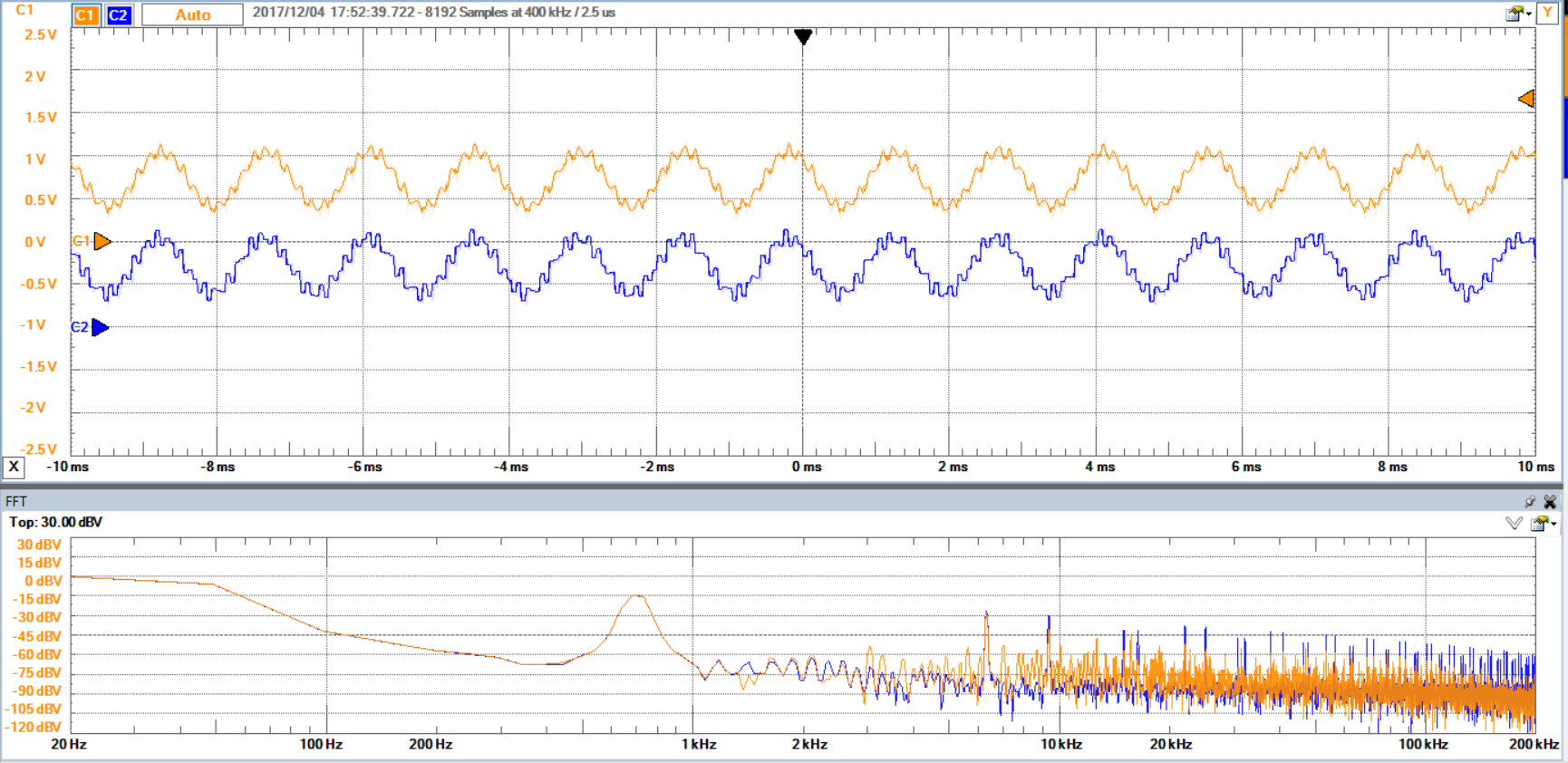
The FFT function is proven functional and quick as advertised. The algorithm was capable of completing a 16 point FFT in less than 64 us. However, filter theory does not appear as simple and functional as originally thought. Additionally, many small errors caused substantial challenges to the lab, such as the filter array being filled incorrectly due to parenthetical errors, or sampling the signal too quickly resulting in a near perfect representation, but mostly unfilterable. We are unsure as to why the filters were not functional, and therefore do not have suggested solutions to attempt.



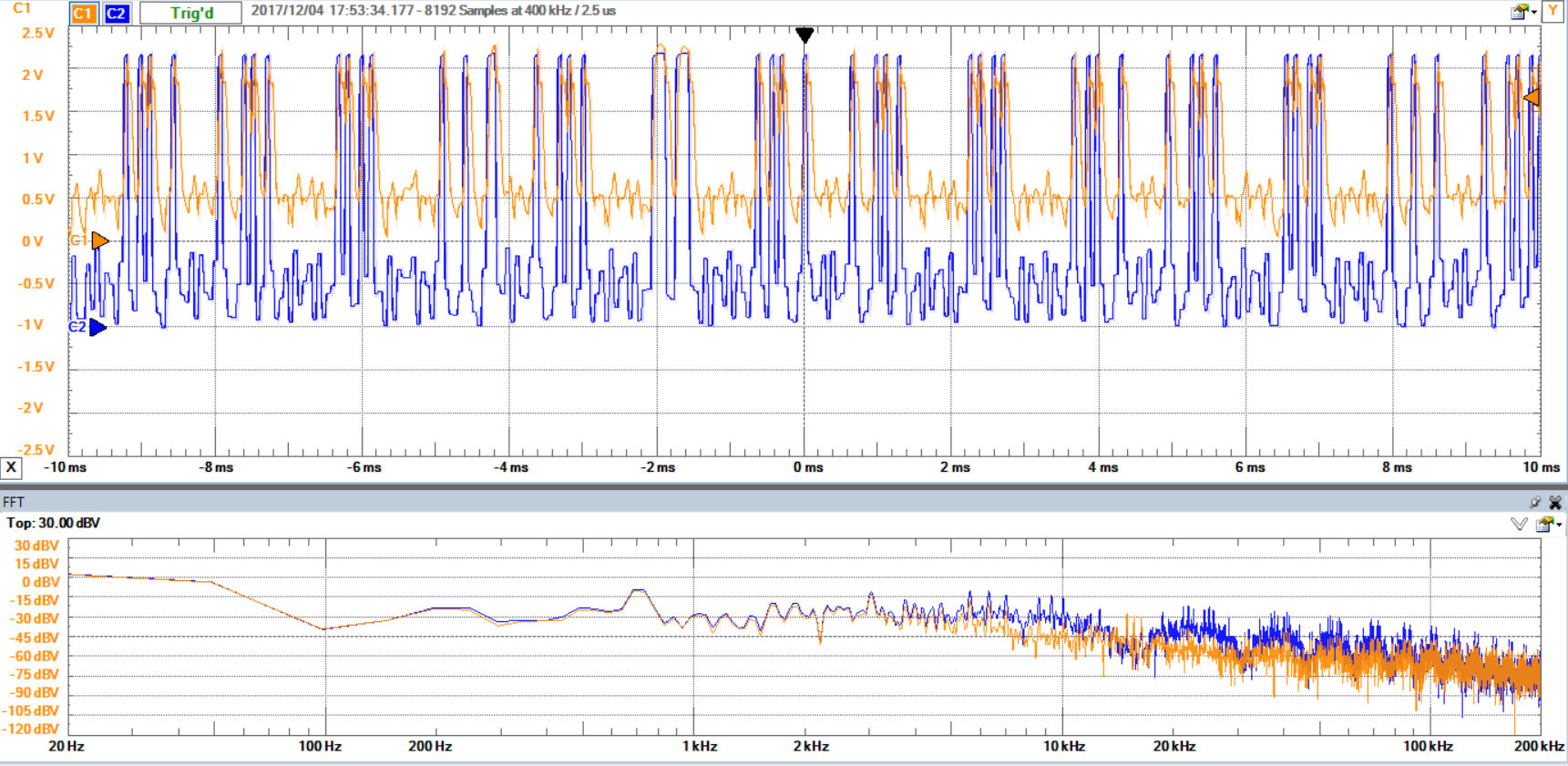
**Figure 1: Input Signal w/ Frequency Components at 700 Hz and 6.2 kHz**



**Figure 2: Reconstructed Output Waveform Reading 1 and Performing the FFT Each Interrupt**



**Figure 3: Reconstructed Low Pass Filtered Output Waveform**



**Figure 4: Reconstructed High Pass Filtered Output Waveform**

**Code:**

// File: MovingAvgFilter.c

//

// Processor: PIC32MX79F512L

// Board: Max32

// Compiler: XC32

// IDE: MPLAB-X

// Date: January 22, 2017

// Status: in development: correct in initial tests

//

// Description:

// Moving Average Filter

// Timer 2 generates interrupts at 15625 i.e. every 64 us

// The ISR reads a sample from the ADC, performs a moving average, and

// writes the result to the 12-bit SPI DAC

// Port B bit 2 is set as digital output to time execution of filter

// Illustrates how to

// 1. set oscillator

// 2. use SPI port and MCP 4822 12-bit DAC

// 3. use timer

// 4. use interrupts

// 5. use on-chip 10-bit ADC

// 6. perform a moving average filter algorithm

// 7. time execution of filter algorithm using output bit and oscilloscope

//

//

// Reference: Chapter 15 of Smith textbook

//

//----------------------------------------------------------------------

// INCLUDES

//----------------------------------------------------------------------

#include <xc.h>

#include <stdio.h>

#include <plib.h>

#include <inttypes.h>

#include <math.h>

#include "ADC.h"

#include "BASICSHIELD.h"

#include "SPI.h"

#include "SYS.h"

#include <dsplib\_dsp.h>

#include <fftc.h>

#define FILTER\_SIZE 11

#define min\_change 200

#define fftc fft16c16 // Number of points in the moving average filter

#define T (0.000064) // Sample interval in seconds

#define Fs (1/T) // Sample frequency

// For blinking LED

#define HALF\_SECOND (0.5/T) // 0.5 seconds / T = counts to reach 1/2 second

#define log2N 4

#define N ( 1 << log2N )

int16c Time[N], Freq[N], scratch[N], Filter[N], Tout[N];

float FilterCoeffFloat = 1.0/FILTER\_SIZE;

/\*float ynFloat;

int Xoldest = 0;

int Xnewest = FILTER\_SIZE - 1;

\*/

unsigned int count = 0; // Used to blink the LED on Port A bit 3 of the MAX32

// -----------------------------------------------------------

// build\_filter\_kernel

// -----------------------------------------------------------

void build\_filter\_kernel(void)

{

//int i = 32767;

int j = 0;

int filter\_size = 3;

//for ( ; j < 16 ; ++j , i -= (32767 >> 4))

//{

// Filter[j].re = i;

//Filter[j].im = 0;

//}

for( ; j < filter\_size; ++j )

{

Filter[j].re = 0;

}

for ( ; j < (N - filter\_size + 1); ++j )

{

Filter[j].re = 1;

}

for ( ; j < N; ++j )

{

Filter[j].re = 0;

}

}

// -----------------------------------------------------------

// multiply\_ffts

// -----------------------------------------------------------

void multiply\_ffts(void)

{

int i = 0;

for( ; i < N; ++i )

{

Freq[i].re \*= Filter[i].re;

Freq[i].im \*= Filter[i].re;

}

}

// -----------------------------------------------------------

// inverse\_fft

// -----------------------------------------------------------

void inverse\_fft(void)

{

int i = 0;

for ( ; i < N; ++i )

{

Freq[i].im \*= -1;

}

mips\_fft16( Tout, Freq, fftc, scratch, log2N );

for( i = 0; i < N; ++i )

{

if( Tout[i].im < 0 )

{

Tout[i].im \*= -1;

}

}

for ( i = 0; i < N; ++i )

{

Tout[i].re = Tout[i].re << ( log2N >> 4 );

Tout[i].im = Tout[i].im << ( log2N >> 4 );

}

Nop();

}

// -----------------------------------------------------------

// Timer2initialize

// -----------------------------------------------------------

// Set for interrupt rate of 15625 Hz

void Timer2initialize(void) {

T2CON = 0x8070; // 80 MHz = 12.5 ns per clock. Prescale = divide by 256

// 256 clocks per count \* 12.5 ns per clock = 3.2 us per count

PR2 = 19; // interrupt after 20 counts = 64 us

}

// -----------------------------------------------------------

// PortsInitialize

// -----------------------------------------------------------

/\*void PortsInitialize(void) {

//\_TRISB2 = 0; // set GPIO bit 2 of port B as output: used to time filter execution

TRISBCLR = 0x04; // Two ways to do the same thing

//\_TRISA3 = 0; // LED connected to Port A bit 3 (pin 13 of MAX32)

TRISACLR = 0x08; // Two ways to do the same thing

}\*/

// -----------------------------------------------------------

// InterruptInitialize

// -----------------------------------------------------------

// Initializes Timer2 interrupt

void InterruptInitialize(void) {

mT2SetIntPriority(1);

INTEnableSystemMultiVectoredInt();

mT2IntEnable(1);

}

// -----------------------------------------------------------

// SystemInitialize

// -----------------------------------------------------------

void SystemInitialize(void) {

SysInitialize();

PortsInitialize();

SPI2initialize();

Timer2initialize();

InterruptInitialize();

//initU1(); // UART

initADC(0xfffc); // Argument determines which Port B bits are analog input (0) and which are digital (1)

// MAX32 pin A0 (Port B bit 0), pin A1 (B1) are analog rest (B2-15) are digital

}

// -----------------------------------------------------------

// Global Variables

// -----------------------------------------------------------

//float xFloat[FILTER\_SIZE + 1];

// -----------------------------------------------------------

// Update Sample Array

// Operates on global variable array and indices Xnewest and Xoldest

// -----------------------------------------------------------

/\*void UpdateSampleArray() {

Xnewest++;

if (Xnewest > FILTER\_SIZE) Xnewest = 0;

Xoldest++;

if (Xoldest > FILTER\_SIZE) Xoldest = 0;

}

int i = 0;

\*/

// -----------------------------------------------------------

// Perform Moving Average Filter

// Performed at a rate of Fs i.e. every T seconds (64 us in this case)

// Reads analog sample from ADC, computes moving average, outputs to DAC

// -----------------------------------------------------------

/\*void AnalogSampleFloatNew(void) {

// i = readADC(1);

static int j = 0;

UpdateSampleArray(); // Update samples

// for( int j = 0; j < N; ++j )

//{

xTime.re[j] = readADC(1) << 5;

xTime.im[j] = 0;

//}

mips\_fft16( xFreq, xTime, fftc, scratch, log2N );

//xFloat[Xnewest] = ((float) i) / 1024.0; // Normalized

// Perform filtering

//SumFloat = SumFloat + xFloat[Xnewest] - xFloat[Xoldest];

//ynFloat = SumFloat\*FilterCoeffFloat;

//i = (int) (ynFloat \* 4096.0); // 12 bits to send to DAC

++j;

if ( j == N )

{

j = 0;

}

// DAC

//////////////////////////// writeDAC\_SPI2(i, 0, 0, 1); //writeDAC\_SPI2(val, channel, gain, shutdown)

}\*/

// -----------------------------------------------------------

// AnalogOutput

// -----------------------------------------------------------

/\*void AnalogOutput(void)

{

writeDAC\_SPI2(xTime.re[i], 0, 0, 1);

}\*/

// -----------------------------------------------------------

// T2InterruptHandler

// -----------------------------------------------------------

void \_\_ISR(\_TIMER\_2\_VECTOR, ipl1) T2InterruptHandler(void) {

// T2 handler code here // used by background cod in main to blink an LED

\_RB2 = 1; // Set Port B bit 2 high to start timing output

//PORTBSET = 0x04; // alternate method

// Read ADC, to filter, output to DAC

Time[N/2].re = readADC(1) << 5;

mips\_fft16( Freq, Time, fftc, scratch, log2N );

multiply\_ffts();

inverse\_fft();

writeDAC\_SPI2( Tout[N/2].re, 0, 0, 1 );

for( count = 0; count < N/2; ++count )

{

Time[count] = Time[count + 1];

}

//AnalogSampleFloatNew(); //

//AnalogOutput();

\_RB2 = 0; // Clear Port B bit 2 to stop timing output

//PORTBCLR = 0x004; // Alternate method

// clear the flag and exit

mT2ClearIntFlag();

} // T2 Interrupt Handler

// -----------------------------------------------------------

// main

// -----------------------------------------------------------

main() {

SystemInitialize();

build\_filter\_kernel();

//for(; i < N; ++i )

//{

// Time[i].re = 0;

//Time[i].im = 0;

//}

while (1) // This loop run except when the timer interrupt occurs

{ // As long as the LEDS change you know the interrupts are occurring

if (count > HALF\_SECOND) {

count = 0;

PORTAINV = 0x08; // Invert Port A bit 3 to blink LED

}

// Uncomment and don't initialize interrupts to test ADC and DAC

//i = readADC(1);

//writeDAC\_SPI2(i, 0, 0, 1);

//printf("%d ", i);

}

} // main