

Lab

MC56F8006DEMO

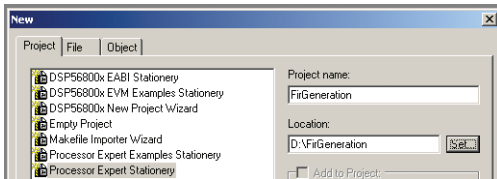
Introduction

This lab is a guide to creating a CodeWarrior™ project to implement a finite impulse response filter (FIR) using Processor Expert™ on the MC56F8006 digital signal controller (DSC).

- A low pass filter is devised to filter out a 2000 Hertz noise signal
- A 1000 Hertz signal is retained
- Sampling rate of signal is 8000 samples per second (SPS)
- Project runs on the MC56F8006DEMO
- Results are graphed by the IDE

Launch CodeWarrior for DSC 8.2.3: Open and Name a New Project

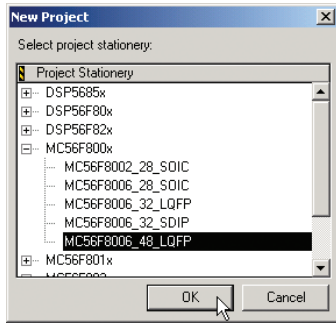
- Open a new CodeWarrior project: **Ctrl+Shift+N**
- Use Processor Expert Stationery and a new project name:



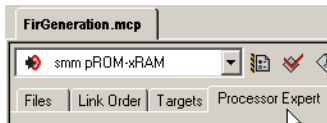
The purpose of this exercise is to get started using the FIR as quickly as possible, using the included filter QEDesign package to obtain verifiable results using the MC56F8006 target. We can assign time values to the sample sequences in an arbitrary manner. Data to be filtered is simply generated by a C program using eight tabled values, and results are viewed using the debugger.

Target MC56F8006_48_LQFP: Select Processor Expert

- Select the MC56F8006 as the target device:



- Select the Processor Expert tab of the project window:



Fs is the sampling frequency. (We will tell the filter design package it is 8000 SPS)
The input consists of the sum of two sines, of zero phase and frequencies $F_s/4$ and $F_s/8$ (2000 and 1000), and of amplitude .45.

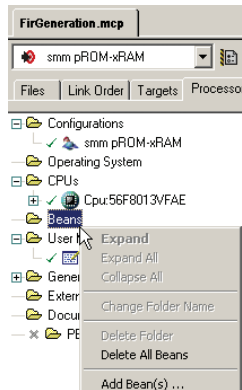
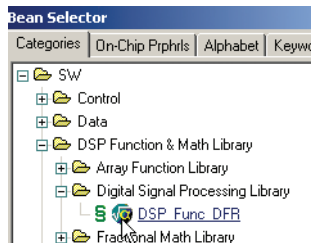
The objective is to design and demonstrate an FIR filter to filter out the higher frequency (a low pass filter), $F_s/4$, using the filter design package provided with CodeWarrior and the FIR bean supplied with PE.

The coefficients generated by the filter design program should be easily importable into our code, and we should not have to count how many coefficients we are working with, having told the filter design tool how many taps we want for the FIR filter.

Add the Processor Expert DSP Func DFR Bean

In the project window:

- Right click on Beans
- Click on Add Bean(s)
- Use Categories Tab

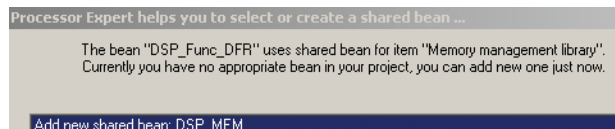


- In Bean Selector window expand SW, DSP Function and Math Library, to select DSP Func DFR, then:

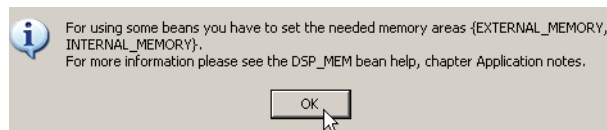


Allow the DSP_MEM Bean Inclusion

- Since dynamic memory is used by default you will see the following window. Click "OK."

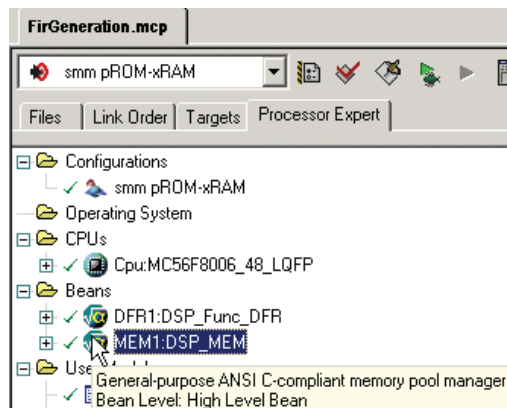


- This memory management library bean is needed for dynamic memory allocation for the FIR
- The FIR bean uses dynamic memory allocation by default
- Static memory allocation may be adopted if needed later
- The following message pops up now:



- Click "OK"
- Next: allocate some memory for the dynamic memory allocation's use

Configure the MEM1:DSP_MEM Bean



- In the project window, double click the MEM1 bean to configure it

Observe the MEM1 Bean's Properties

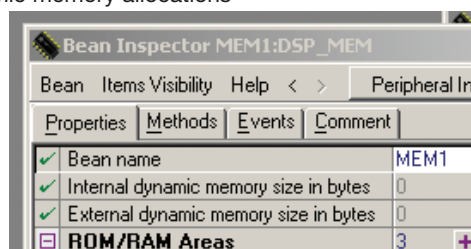
- The internal dynamic memory size is initially zero for this bean
- There is no external memory for this part

Conclusion: Some internal memory must be allocated for dynamic use in this bean

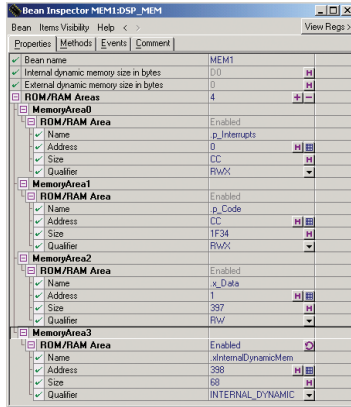
Configure MEM1 Bean for Dynamic Memory Allocation 1/2

- Click once on the + to add a section for dynamic memory allocations

A fourth memory area, MemoryArea3, will be generated. This memory area must be reconfigured. The qualifier must be set to INTERNAL_DYNAMIC. The size cannot be zero. 68 (hexadecimal) can be used as a starting size for this project. Finally, we must resolve the resulting overlap with MemoryArea2. The resulting bean configuration is shown to the right.



Configure MEM1 Bean for Dynamic Memory Allocation 2/2



Name can be whatever user wishes to name it

Since we are using the simulator, we need a source of data to filter. We can easily generate a couple of sine waves and add them together.

We can do this in fixed point with a very small amount of code and without resorting to the use of FILE IO, which can later be added to try filtering recorded live audio samples.

Since we are generating samples without respect to real-time, we can just work in terms of the sampling frequency, F_s . The highest frequency we might try to represent with samples at the rate F_s would be less than $F_s/2$.

Picking $F_s/4$ as one of our frequencies. In this case we have 0,A,0,-A,0,A,0,-A as an easy to generate sine wave of amplitude A.

We can also easily generate the samples for the frequency $F_s/8$. In this case, we have the sample series 0,S,A,S,0,-S,-A,-S (repeat same eight samples add infinitum).

Where $S = A * 1 / \sqrt{2}$

If we add these two series, we have the sum of two sine waves of two different frequencies:

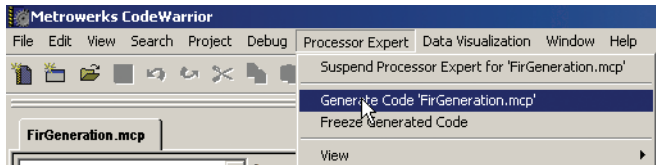
0,A+S,A,S-A,0,A-S,-A,-(A+S) (repeat the same 8 samples add infinitum).

If $A = .45$ then the desired input sample series is the following repeated:

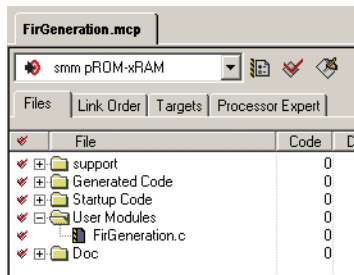
- 0.0,
 $.45 + 0.31819805153394638598037996294718 = 0.76819805153394638598037996294718,$
- .45,
 $-0.131801948466053614019620037053,$
- 0.0,
 $0.131801948466053614019620037053,$
- -.45,
 $-0.76819805153394638598037996294718$

Generate the Code Using Processor Expert

- Generate code 'FirGeneration.mcp'

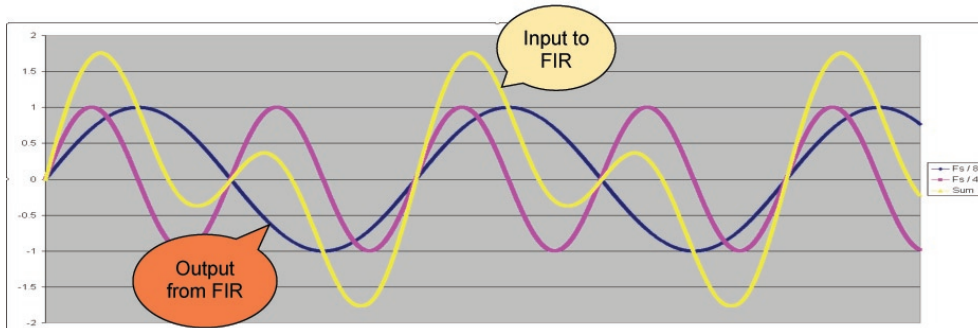


- Once this is done, select the Files tab of the project window to edit FirGeneration.c
- This is the file we will add our code to filter out high-frequency noise with an FIR
- Hint: Expand User Modules to find FirGeneration.c, then double-click it.



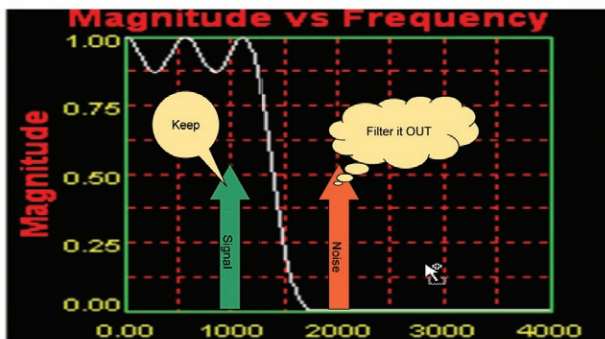
Do not build the project because the filter is not yet included

Generate Canned Input Signal Plus Noise



- The black signal of interest is at $F_s/8$ or 1000 Hertz
- The pink noise is a 2000 Hertz signal
- The yellow resultant sum, our input, needs filtering
- A low pass filter can filter out the 2000 Hertz signal

Signal and Noise in Frequency Domain



Add Global Variables in FirGenerate.c

- FirCoefs.h is generated by the QEDesign tool
- GeneratedSamples is the sum of two sines, 1000 and 2000 Hertz, zero phase, amplitude .45
- FirOutput is the filtered version of the GeneratedSamples input to the FIR

The following are samples from the yellow input signal:

```

Frac16 FirOutput[8] ;

void main(void)
{
    Frac16 * pC;
    dfr16_tFirStruct * pFIR;
    Frac16 * pX;
    Frac16 * pZ;

    /* Write your local variable definition here */

    /** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! */
    PE_low_level_init();
    /** End of Processor Expert internal initialization. */

    /* Write your code here */
    pC = (Frac16 *) FirCoefs ;
    pFIR = DFR1_dfr16FIRCreate( pC, 31);
    pX = (Frac16 *) GeneratedSamples ;
    pZ = FirOutput ;

    for(;;) {
        DFR1_dfr16FIR( pFIR, pX, pZ, 8);
    }
}

```

Fs nominally 8000 SPS

Use the following to copy text for pasting. Not required to enter all the digits.

GeneratedSamples are added to the program. Only eight values are needed to be stored in the table, since the series repeats after eight samples.

// use const to output coefficients to the data section to be placed in data memory.

const

#include "FirCoefs.h"

// Data samples represent sum of two sine waves with zero phase.

// Fs/4, and Fs/8 are the frequencies of the two. They are of

// equal amplitude, 0.45.

// Sample zero is phase zero for both sine's.

const Frac16 GeneratedSamples[] = {

FRAC16(0.0),

FRAC16(0.76819805153394638598037996294718),

FRAC16(0.45),

FRAC16(-0.131801948466053614019620037053),

FRAC16(0.0),

FRAC16(0.131801948466053614019620037053),

FRAC16(-.45),

FRAC16(-0.76819805153394638598037996294718)

};

Frac16 FirOutput[8] ;

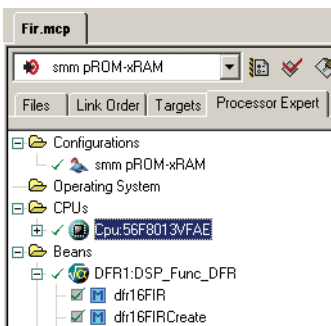
Copy and Paste Text Prior to Main()

// use const to output coefficients to the data section to be placed in data memory.

```
const
#include "FirCoefs.h"
// Data samples represent sum of two sine waves with zero phase.
// Fs/4, and Fs/8 are the frequencies of the two. They are of
// equal amplitude, 0.45.
// Sample zero is phase zero for both sine's.
const Frac16 GeneratedSamples[] = {
    FRAC16(0.0),
    FRAC16(0.76819805153394638598037996294718),
    FRAC16(0.45),
    FRAC16(-0.131801948466053614019620037053),
    FRAC16(0.0),
    FRAC16(0.131801948466053614019620037053),
    FRAC16(-.45),
    FRAC16(-0.76819805153394638598037996294718)
};
Frac16 FirOutput[8];
```

Drag'n'drop the FIR Methods into Main()

- Select Processor Expert tab in FirGeneration project window
- Expand the DFR1:DSP_Func_DFR bean
- DFR1_dfr16FIR(dfr16_tFirStruct *pFIR, Frac16 *pX, Frac16 *pZ, UInt16 n);
- EXPORT dfr16_tFirStruct * DFR1_dfr16FIRCreate(Frac16 *pC, UInt16 n)
- This example was created by using the Drag'n'drop method declaration
- To enable it, configure Processor Expert Environment Options
- The prototypical variable names were used in our program
- The definitions were created by copying and pasting from these lines



Finish Writing Main() Function for FirGenerate

```
#include "IO_Map.h"
// use const to output coefficients to the data section to be placed in data memory.
const
#include "FirCoefs.h"
// Data samples represent sum of two sine waves with zero phase.
// Fs/4, and Fs/8 are the frequencies of the two. They are of
// equal amplitude, 0.45.
// Sample zero is phase zero for both sines.
const Frac16 GeneratedSamples[] = {
    FRAC16(0.0),
    FRAC16(0.76819805153394638598037996294718),
    FRAC16(0.45),
    FRAC16(-0.131801948466053614019620037053),
    FRAC16(0.0),
    FRAC16(0.131801948466053614019620037053),
    FRAC16(-.45),
    FRAC16(-0.76819805153394638598037996294718)
};
Frac16 FirOutput[8];

void main(void)
{
    //
```

- Right click to add eventpoints for audio and visual
- Use next page to copy text for pasting
- Make sure to set the visual break point as shown
- Set an audio breakpoint as shown

Note the V for Visual, and the speaker icon for audio. The visualization is updated when the V breakpoint is reached. The audio breakpoint will play a sound and delay the visual updates so that each can be noticed.

Copy and Paste Text to be Main() Function

```

void main(void)
{
    Frac16 * pC;
    dfr16_tFirStruct * pFIR;
    Frac16 * pX;
    Frac16 * pZ;

    /* Write your local variable definition here */

    /** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! */
    PE_low_level_init();
    /** End of Processor Expert internal initialization.                */

    /* Write your code here */
    pC = (Frac16 *) FirCoefs ;
    pFIR = DFR1_dfr16FIRCreate( pC, 31);
    pX = (Frac16 *) GeneratedSamples ;
    pZ = FirOutput ;

    for(;;) {
        DFR1_dfr16FIR( pFIR, pX, pZ, 8);
    }
}

```

Alternative Fast Response Implementation

Changing to process one sample per call can allow for faster response in tight control loop applications such as found in motor control. Hint: All changes in red.

```
Frac16 FirOutput[1]; // was 8
```

```

void main(void)
{
    Frac16 * pC;
    dfr16_tFirStruct * pFIR;
    Frac16 * pX;
    Frac16 * pZ;
    int i ;

    /* Write your local variable definition here */

    /** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! */
    PE_low_level_init();
    /** End of Processor Expert internal initialization.                */

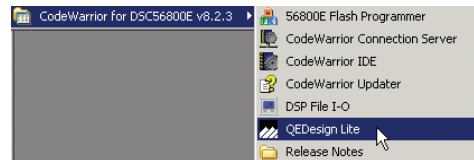
    /* Write your code here */
    pC = (Frac16 *) FirCoefs ;
    pFIR = DFR1_dfr16FIRCreate( pC, 31);
    pX = (Frac16 *) GeneratedSamples ;
    pZ = FirOutput ;

    for(;;)
    {
        for (i=0 ; i<8 ; i++)
        {
            pX = (Frac16 *) andGeneratedSamples[i];
            DFR1_dfr16FIR( pFIR, pX, pZ, 1); // was 8
        }
    }
}

```


Design a Filter: Free Filter Design Tool

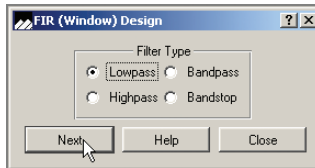
- Launch QED Filter Design Package



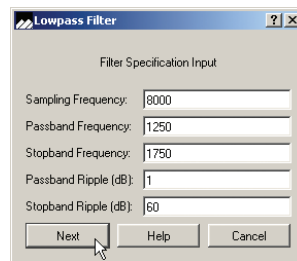
- Select Equiripple FIR Design



- Lowpass



- Input filter parameters, select Next, get 31 taps, select Next
- Only need 31 taps because of only 60 dB stopband ripple



- Use the output of the filter design package, QEDesign Lite for 56800/E. Run the program from the same menu you selected the CodeWarrior compiler from. Hit the EQ button. This is for equiripple FIR. Then, select lowpass. Use F_s of 8000, band pass of 1250, band stop of 1750. This will give a 500 Hertz transition band. Use 1 dB pass band ripple, and 60 dB of stop band ripple. Use the suggested number of taps, only 31 in this case. You can experiment with smaller passband ripples to increase the number of taps.

Equiripple (Parks McClellan) FIRs

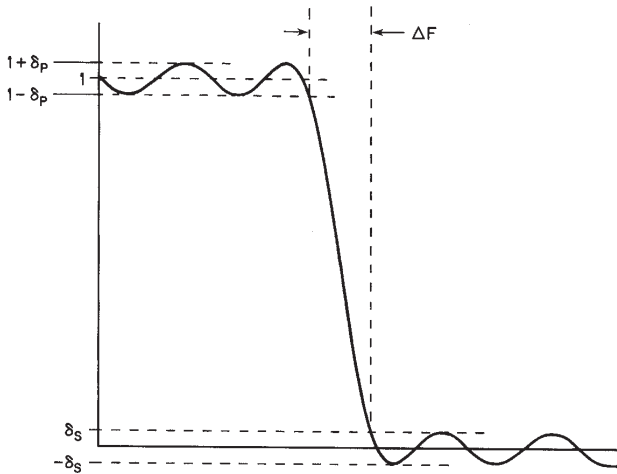
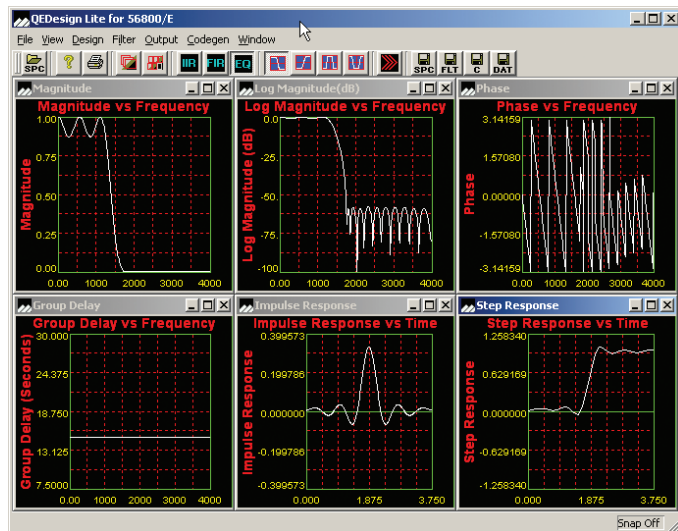


Figure 14.1 Typical amplitude response of an FIR approximation to an ideal lowpass filter.

- Clearly the most popular approach to FIRs
 - Supports arbitrary band shape
 - Filters have equiripple in each passband or stopband
- Parks McClellan is the most popular way of designing general purpose FIR filters
- Besides passbands and stopbands, you specify the amount of ripple you can tolerate in the passbands and the stopbands. (Ripple in the stopbands is equivalent to attenuation.) It is called “equiripple” because there are equal numbers of plus ripples as negative ripples. (In this case two of each in both the passband and stopband.)

Filter Design: Graphic Results

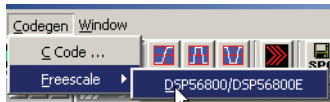
Here's what you should see:



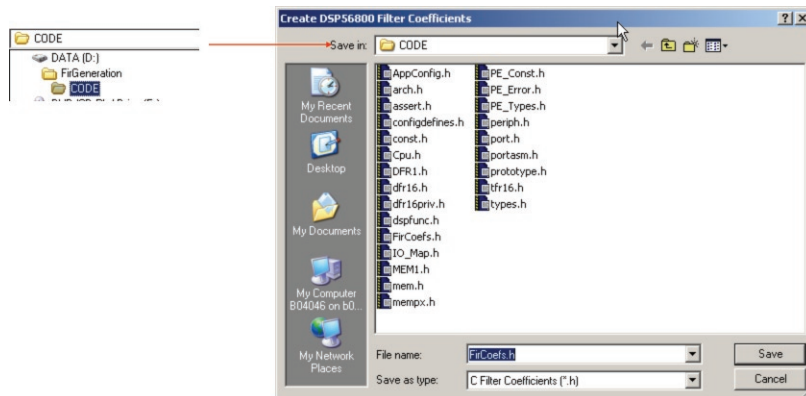
- We can save the coefficients (taps) generated into a file format that can be used by a CodeWarrior/Processor Expert project

Filter Design: Save the Coefficients

- Save the coefficients in the CODE directory



File Name: FirCoefs.h



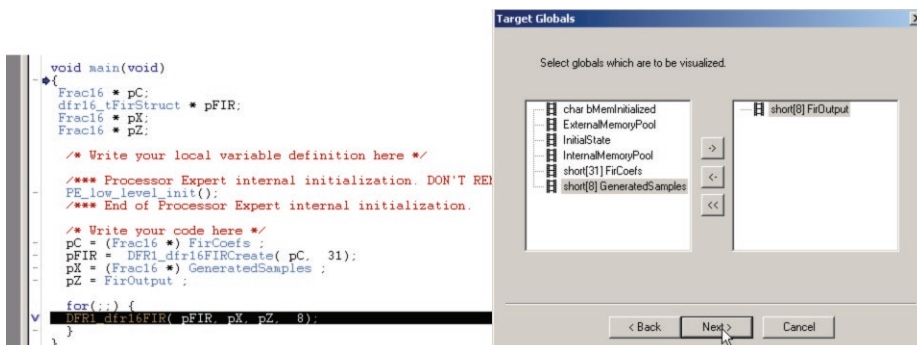
- We generate the coefficients header file and then rebuild the project to include these new coefficients
- We download the new project onto the demo board, run it, and use SweepGen to generate a 200–2000 Hz FM chirp. We can turn the filter on and off and see its effect on the FFT bin LEDs.

Run Program: It Breaks at Main for Debugging

- At breakpoint

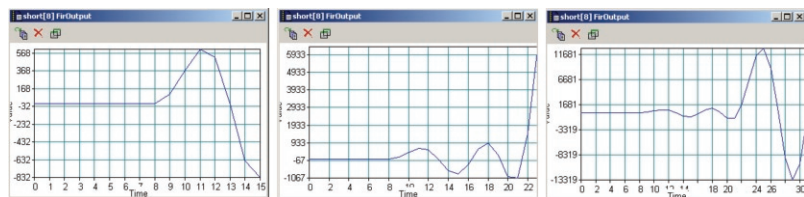


- Select Variables, Next, short(8) FirOutput, ->, Next

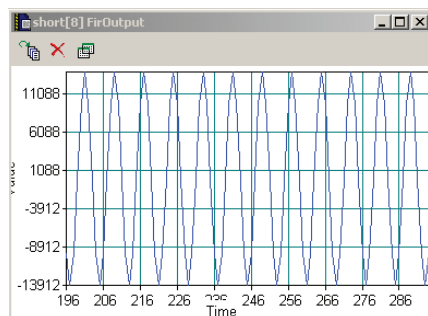


- After hitting the break at main, run again, then see the results on the next page

Visual Output Showing 1000 Hertz Only



The FIR in this case is set up to process eight samples at a time. Above is transient response.



Steady state response. 1000 Hertz lowpass filter is filtering out the 2000 Hertz signal, keeping the 1000. Thanks to the breakpoints, the visual data is presented at a pace where each frame can be observed.