IMPLEMENTING RESPONSE OF CHEBYSHEV FIR FILTER ON HARDWARE USING ADSP-2181

BY:

Dibya Gyan Murti	031033
Kumar Kunal	031047
Vikash Kumar	031057
Vineet Anand	031086

विद्या तत्व ज्योतिसम:

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY

May 2007

Submitted in partial fulfillment of the Degree of Bachelor of Technology

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY-WAKNAGHAT

CERTIFICATE

This is to certify that the work entitled, **"Implementing response of chebyshev fir fllter on Hardware using ADSP2181"** submitted by **Dibya Gyan Murti, Kumar Kunal, Vikash Kumar, Vineet Anand** in partial fulfillment for the award of degree of Bachelor of Technology in may 2007 of Jaypee University of Information Technology has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

[illegible]

Mr.Vinay Kumar
Senior Lecturer
Department Of Electronics and
Communication Engineering

[illegible]

Dr. S.V.Bhooshan
HOD, Department of
Electronics and
Communication Engineering

ACKNOWLEDGEMENTS

We would like to express our deep sense of gratitude and our heartly thanks to our faculty "Mr. Vinay Kumar", Senior Lecturer and "Dr.S.V.Bhooshan", HOD Department of Electronics and communication Engineering, Jaypee University of Information Technology for his painstaking effort and spending his valuable and precious time in guiding us through the progress of project **"Implementing response of chebyshev fir filter on Hardware using ADSP2181"**.

We would also like to thank "Mr.Varun Aggarwal", Computer Science Lab, Massachusetts Institute of Technology Cambridge, MA for guiding us through email in setting the framework of the project

➢ Dibya Gyan Murti	031033
➢ Kumar Kunal	031047
➢ Vikash Kumar	031057
➢ Vineet Anand	031086

TABLE OF CONTENTS

List of Figures:-

ABSTRACT

This project takes an analytical and practical approach towards constructing and implementing Finite Impulse Response (FIR) filters designs on a ADSP-2181 system. We have found that EZ-KIT LITE that interfaces with computer and EZ-ICE emulator are useful for DSP applications Such as Speech, data, image and signals.

This project also deals with some of the techniques used to design FIR filters. Different optimization techniques involved in FIR filter design are also covered for FIR filter design. These optimization techniques reduce the error caused by frequency sampling technique at the non-sampled frequency points

Introduction:

A filter is a linear frequency selective circuit, which attenuates certain frequencies and amplifies certain others. An ideal (low pass) filter with so-called "brick wall" characteristic has frequency domain characteristics. The primary performance criterion of a filter is its frequency domain magnitude response (henceforth called, magnitude response). A second set of performance criteria are expressed in time domain characteristics.

Digital Signal Processing (DSP) functionality is difficult to implement in an all-analog system. Furthermore, it provides a medium of design that is highly modifiable and customizable. Adding a DSP component to an all analog system allows the designer to rapidly prototype, test and implement many new and powerful algorithms. Also, once the DSP infrastructure has been installed further revisions can be implemented at a pace much quicker than in all-analog. And ADSP-2181 is one of the most powerful DSP processors which we here are going to use.

CHAPTER-1

Filters

1.1 What do we mean by filters:-

Filters are implemented by combining signals with their delayed copies, in different ways. Two basic families of filters exist:

(1) Filters that use delayed copies of their input, called feed forward or finite impulse response (FIR).

(2) Filters that use delayed outputs, called feedback or infinite impulse response (IIR). They can also include feed forward elements. Resonators and Butterworth filters are IIRs.

In some digital filters, delays will be as small as 1 sample. The order of the filter is determined by the order of the delay used. If a filter uses a max of 2-sample delays, it is classed as 2nd order.

Filters are defined by their equations. These will show:

(a) The delays used in the filter.

(b) The coefficients (gain multipliers) associated with each delay.

Example:

$y(n) = ax(n) + a1x(n-2) - b1y(n-1) - b2y(n-2)$

n is the current sample of a signal; y(n) is the output and x(n) the input. a1 is the coefficient associated with a 2-sample delay of the input and b1 / b2 are the coefficients associated with 1- and 2-sample delays of the filter output.

The frequency response is how a filter alters an input signal, in terms of its amplitude and phase at different frequencies. The amplitude response is the part of the frequency response that has to do with boosting or attenuating the different input frequencies.

The phase response determines the timing delays imposed on different frequencies. It can be linear (the same phase change at all frequencies) or non-linear. From the filter equation (and coefficients) we can determine the frequency response, and vice-versa.

One of the simplest filters we can describe is the 1st order feed forward (FIR) averaging filter:

y(x) = (x(n) + x(n-1))*0.5

As we can see from the filter equation, all it does is to take the average between the current sample and the previous sample and output it. This filter will exhibit a low-pass amplitude response and have a linear phase delay of ½ samples. Its amplitude and phase response are characterized as:

$$A(freq) = \cos(\pi freq/sr)$$

$$\theta(freq) = -\pi \times freq / sr$$

FIR filters can be designed to have linear phase response (as in the case above), whereas IIR filters will always exhibit non-linear phase responses. In summary:

IIR filters are generally simple to implement, but complex to design. One way round this is that they are generally offered in "pre-packed" formats, ie. Resonators, Butterworth, elliptical, etc. Their cutoff frequencies and bandwidths can be made time-varying, which is an important characteristic for musical applications.

FIR filters are simpler to design and can offer linear-phase characteristics, so they are often preferred by engineers. Implementing them is also easy, but depending on the order of the filter, they can be computationally intensive and are not time-varying.

1.2 Why filters?

- The amplitude response (plotted in terms of amplitude vs. frequency)
- The phase response (plotted in terms of phase vs. frequency)

What does phase response mean?

- Linear phase (i.e. constant time delay)
- Minimum phase (all pole filter)
- Non-minimum-phase (poles and zeros) Linear phase is an important subset of this class that has all zeros.

We are talking about single filters here, not filter banks. That is another subject, and one that places more constraints on individual filters

There are other tradeoffs possible:

IIR filters can have:

Pass band ripple only

Stop band ripple only

Neither pass band nor stop band ripple (monotonic response)

Both pass band and stop band ripple

FIR filters as usually designed can have:

Ratio of pass band ripple to stop band ripple controlled via design parameters. The filter response is not defined in a "transition" band.

There are other FIR types possible; they are not that common in most present-day uses.

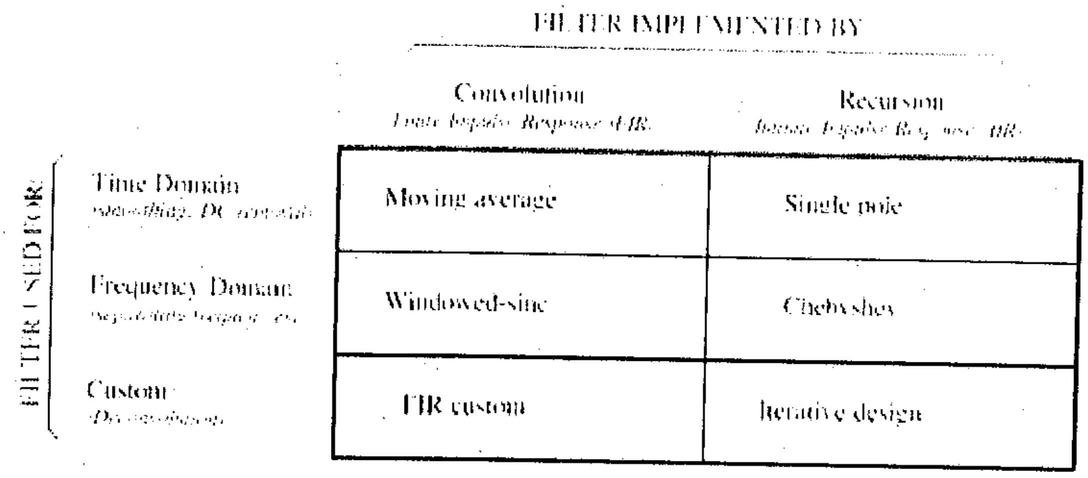
The Cutoff Frequency:

The boundary line in question is somewhat arbitrary, because there is no clear frequency such that all signals above it are passed intact, while all signals below that frequency are entirely blocked. Rather, there will be a "transition zone," or range of frequencies, over which the inputs signal, will be partially transmitted to the output.

Nevertheless, we must select some specific frequency such that we can say (for the high-pass filter) that all signals above this frequency will be passed without appreciable loss, while all signals below this frequency will be blocked to a significant extent. This frequency will be designated as the cutoff frequency (often designated f_{CO}) for our filter.

We have already noted that for very high frequencies there will be no appreciable voltage drop across the capacitor, while for very low frequencies there will be no appreciable voltage drop across the resistor. The transition zone, then, must be in between these extremes, where some of the signal voltage will be dropped across each of the components. And the logical place to start is to set the cutoff frequency at the point where the voltage drops across the two components are the same. The cutoff frequency is also the frequency at which half of the power in the input signal is absorbed by the filter, and only the other half makes it to the output. Therefore, it is sometimes known as the half power frequency, although that designation is no longer used as much as it was in the past.

1.2 Filter Classification:

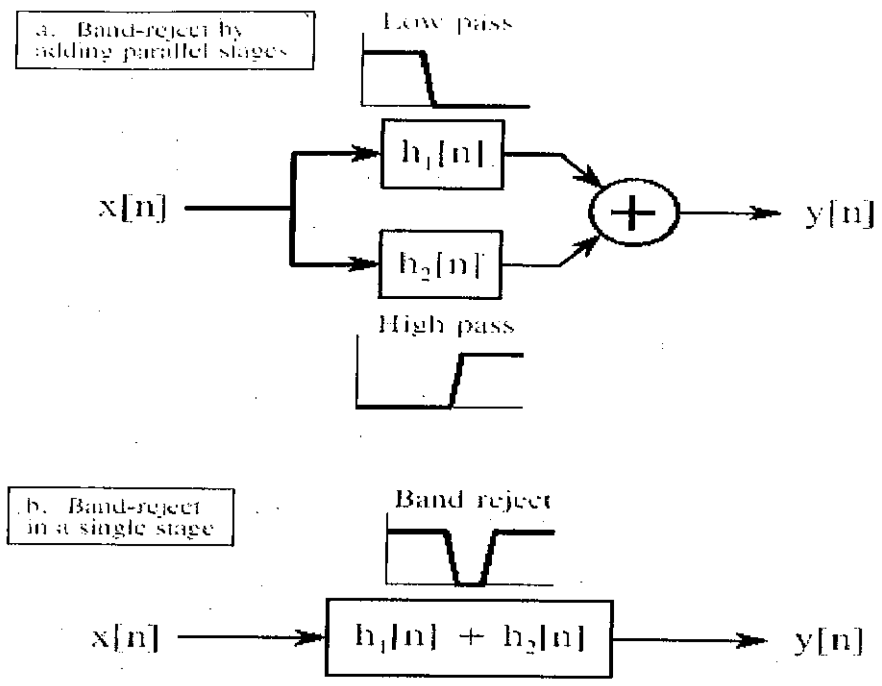


FILTER USED FOR:	FILTER IMPLEMENTED BY: Convolution *(Finite Impulse Response (FIR))*	Recursion *(Infinite Impulse Response (IIR))*
Time Domain *(smoothing, DC removal)*	Moving average	Single pole
Frequency Domain *(separating bands of frequencies)*	Windowed-sinc	Chebyshev
Custom *(Deconvolution)*	FIR custom	Iterative design

Filter classification. Filters can be divided by their use, and how they are implemented

Above table summarizes how digital filters are classified by their use and by their implementation. The use of a digital filter can be broken into three categories: time domain, frequency domain and custom. As previously described, time domain filters are used when the information is encoded in the shape of the signal's waveform.

Time domain filtering is used for such actions as: smoothing, DC removal, waveform shaping, etc. In contrast, frequency domain filters are used when the information is contained in the amplitude, frequency, and phase of the component sinusoids. The goal of these filters is to separate one band of frequencies from another. Custom filters are used when a special action is required by the filter, something more elaborate than the four basic responses (high-pass, low-pass, band-pass and band-reject).

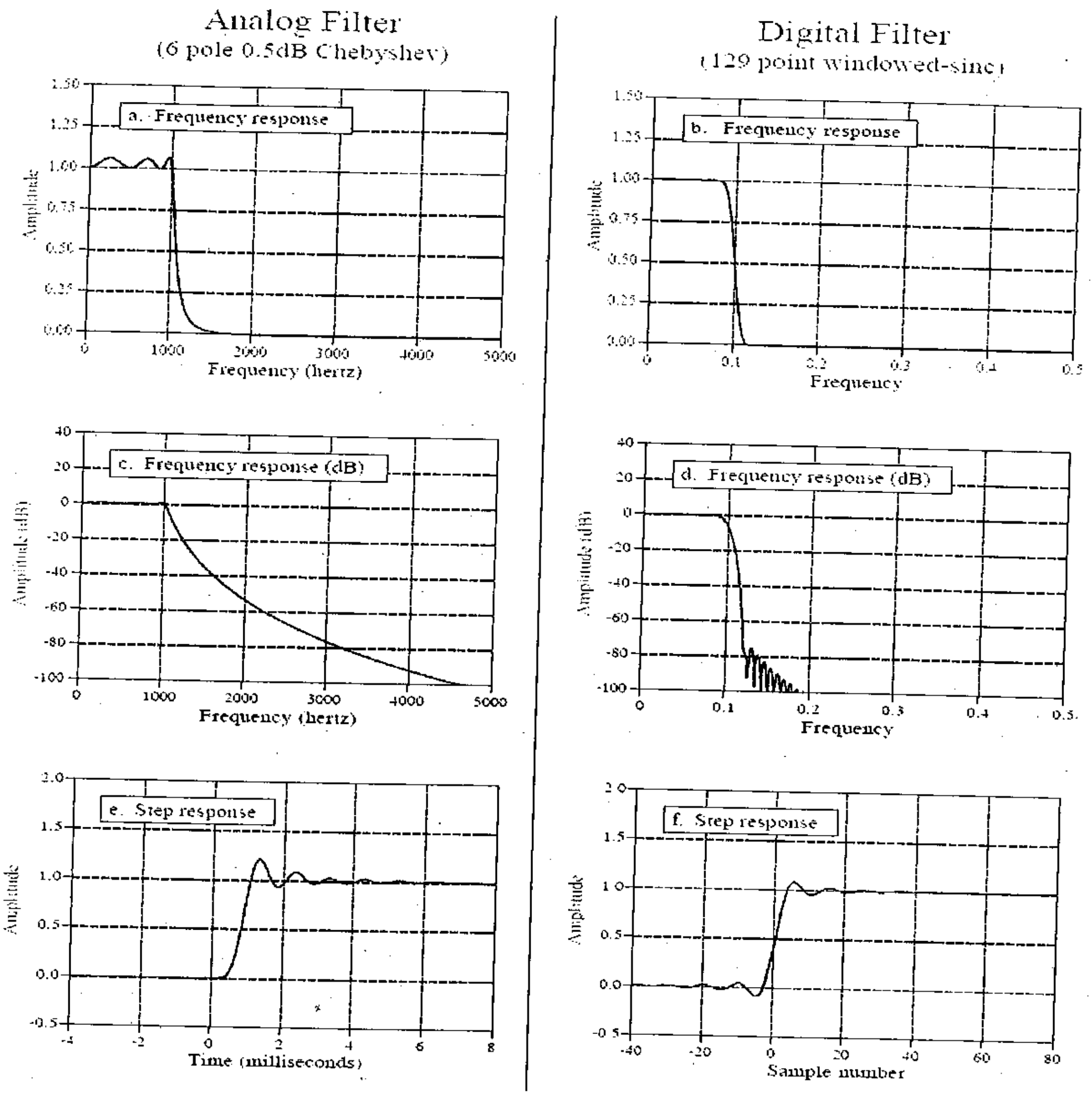

a. Band-reject by adding parallel stages
Low pass
x[n]
h1[n]
h2[n]
y[n]
High pass
b. Band-reject in a single stage
Band reject
x[n]
h1[n] + h2[n]
y[n]

Designing a band-reject filter . As shown in (a), a band-reject filter is formed by the parallel combination of a low-pass filter and a high-pass filter with their outputs added. Figure (b) shows this reduced to a single stage, with the filter kernel found by adding the low-pass and high-pass filter kernels.

Digital filters can be implemented in two ways, by convolution (also called finite impulse response or FIR) and by recursion (also called infinite impulse response or IIR). Filters carried out by convolution can have far better performance than filters using recursion, but execute much more slowly.

1.4 Analog vs. Digital Filters:

Most digital signals originate in analog electronics. If the signal needs to be filtered, is it better to use an analog filter before digitization, or a digital filter after? We will answer this question by letting two of the best contenders deliver their blows.



Comparison of analog and digital filters . Digital filters have better performance in many areas, such as: pass band ripple, (a) vs. (b), roll-off and stop band attenuation, (c) vs. (d), and step response symmetry, (e) vs. (f). The digital filter in this example has a cutoff frequency of 0.1 of the 10 kHz sampling rate. This provides a fair comparison to the 1 kHz cutoff frequency of the analog filter.

The goal will be to provide a low-pass filter at 1 kHz. Fighting for the analog side is a six pole Chebyshev filter with 0.5 dB (6%) ripple. This can be constructed with 3 op amps, 12 resistors, and 6 capacitors. In the digital corner, the windowed-sinc is warming up and ready to fight. The analog signal is digitized at a 10 kHz sampling rate, making the cutoff frequency 0.1 on the digital frequency scale. The length of the windowed-sinc will be chosen to be 129 points, providing the same 90% to 10% roll-off as the analog filter. Fair is fair. Let's compare the two filters blow-by-blow. As shown in figure (a) and (b), the analog filter has a 6% ripple in the pass band, while the digital filter is perfectly flat (within 0.02%). The analog designer might argue that the ripple can be selected in the design; however, this misses the point. The flatness achievable with analog filters is limited by the accuracy of their resistors and Most digital signals originate in analog electronics. The goal will be to provide a low-pass filter at 1 kHz. Fighting for the analog side is a six pole Chebyshev filter with 0.5 dB (6%) ripple. In the digital corner, the windowed-sinc is warming up and ready to fight. The analog signal is digitized at a 10 kHz sampling rate, making the cutoff frequency 0.1 on the digital frequency scale. The length of the windowed-sinc will be chosen to be 129 points, providing the same 90% to 10% roll-off as the analog filter.

Let's compare the two filters blow-by-blow. As shown in (a) and (b), the analog filter has a 6% ripple in the pass band, while the digital filter is perfectly flat (within 0.02%). The analog designer might argue that the ripple can be selected in the design; however, this misses the point. The flatness achievable with analog filters is limited by the accuracy of their resistors and Even if a Butterworth response is designed (i.e., 0% ripple), filters of this complexity will have a residue ripple of, perhaps, 1%. On the other hand, the flatness of digital filters is primarily limited by round-off error, making them hundreds of times flatter than their analog counterparts. Score one point for the digital filter.

Next, now take a look at the frequency response on a log scale, as shown in (c) and (d). Again, the digital filter is clearly the victor in both roll-off and stop band attenuation. Even if the analog performance is improved by adding additional stages, it still can't compare to the digital filter. For instance, imagine that you need to improve these two parameters by a factor of 100. This can be done with simple modifications to the windowed-sinc, but is virtually impossible for the analog circuit. Score two more for the digital filter. The step response of the two filters is shown in (e) and (f). The digital filter's step response is symmetrical between the lower and upper portions of the step, i.e., it has a linear phase. The analog filter's step response is not symmetrical, i.e., it has a nonlinear phase. One more point for the digital filter. Lastly, the analog filter overshoots about 20% on one side of the step. The digital filter overshoots about 10%, but on both sides of the step. Since both are bad, no points are awarded.

In spite of this beating, there are still many applications where analog filters should, or must, be used. This is not related to the actual performance of the filter (i.e., what goes in and what comes out), but to the general advantages that analog circuits have over digital techniques. The first advantage is speed: digital is slow; analog is fast. Even simple op amps can operate at 100 kHz to 1 MHz, 10 to 100 times as fast as the digital system!

The second inherent advantage of analog over digital is dynamic range. This comes in two flavors. Amplitude dynamic range is the ratio between the largest signal that can be passed through a system, and the inherent noise of the system. For instance, a 12 bit ADC has a saturation level of 4095, and an rms quantization noise of 0.29 digital numbers, for a dynamic range of about 14000. In comparison, a standard op amp has a saturation voltage of about 20 volts and an internal noise of about 2 microvolts, for a dynamic range of about ten million. Just as before, a simple op amp devastates the digital System.

The other flavor is frequency dynamic range. For example, it is easy to design an op amp circuit to simultaneously handle frequencies between 0.01 Hz and 100 kHz (seven decades). When this is tried with a digital system, the computer becomes swamped with data. For instance, sampling at 200 kHz, it takes 20 million points to capture one complete cycle at 0.01 Hz. we may have noticed that the frequency response of digital filters is almost always plotted on a linear frequency scale, while analog filters are usually displayed with a logarithmic frequency. This is because digital filters need a linear scale to show their exceptional filter performance, while analog filters need the logarithmic scale to show their huge dynamic range.

CHAPTER-2

DIGITAL FILTERS

2.1 Introduction to Digital filters:

Digital filters are used for two general purposes: (1) separation of signals that have been combined, and (2) restoration of signals that have been distorted in some way. Analog (electronic) filters can be used for these same tasks; however, digital filters can achieve far superior results.

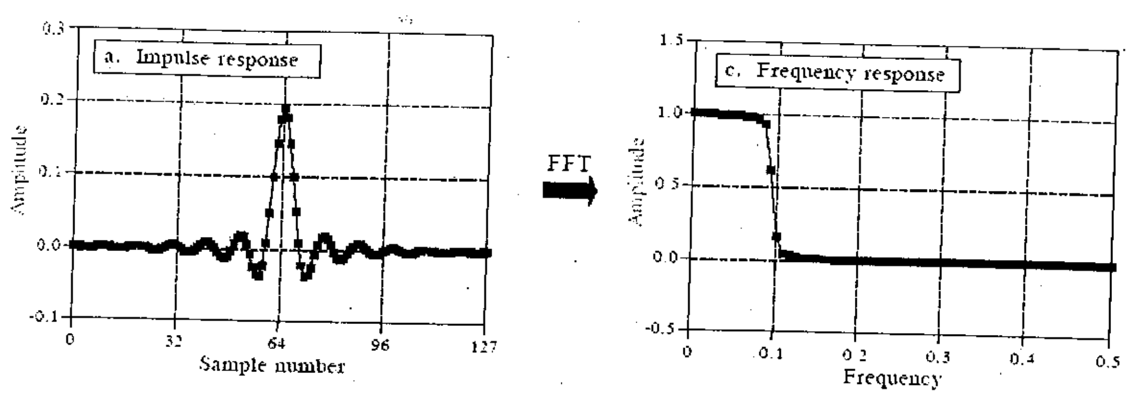
Digital filters are a very important part of DSP. In fact, their extraordinary performance is one of the key reasons that DSP has become so popular. As mentioned in the introduction, filters have two uses: signal separation and signal restoration. Signal separation is needed when a signal has been contaminated with interference, noise, or other signals. For example, imagine a device for measuring the electrical activity of a baby's heart while still in the womb. The raw signal will likely be corrupted by the breathing and heartbeat of the mother. A filter might be used to separate these signals so that they can be individually analyzed. Signal restoration is used when a signal has been distorted in some way. For example, an audio recording made with poor equipment may be filtered to better represent the sound as it actually occurred. Another example is the deblurring of an image acquired with an improperly focused lens, or a shaky camera.

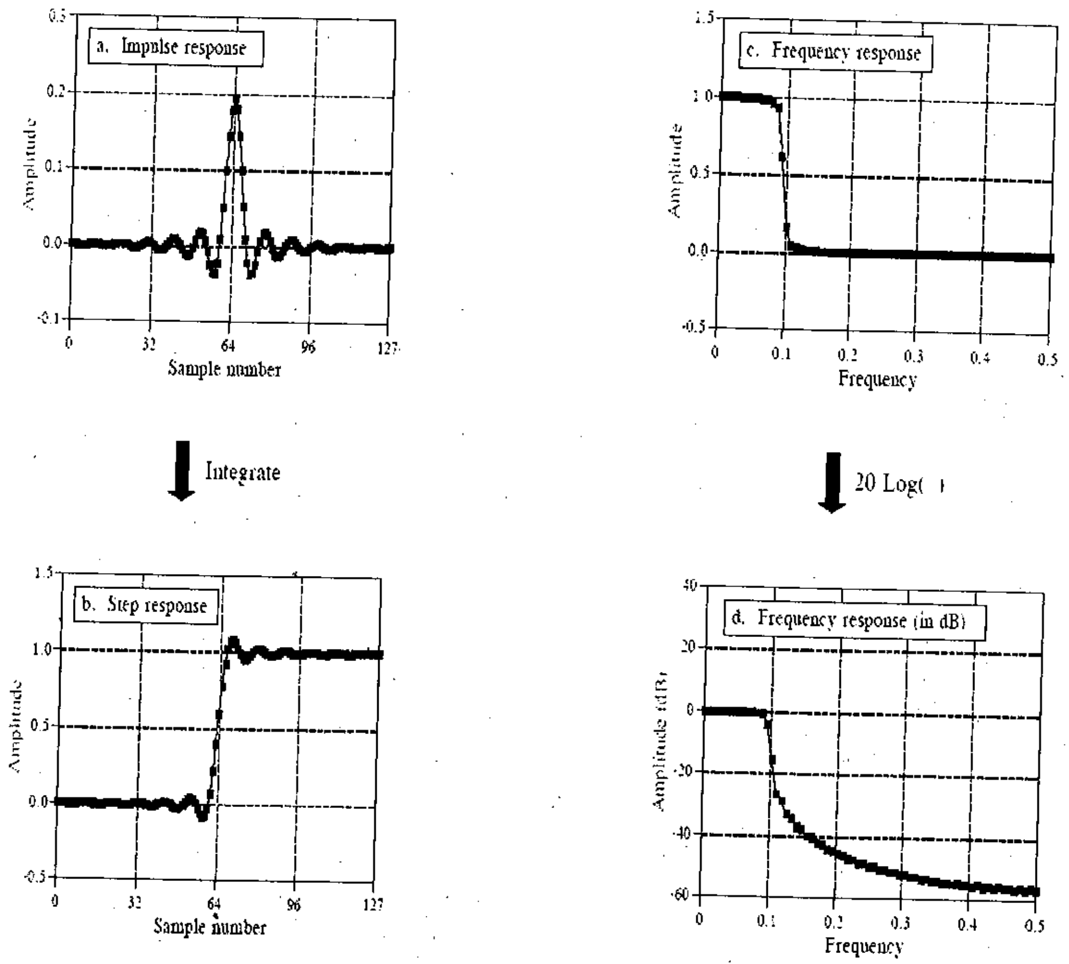
These problems can be attacked with either analog or digital filters. Analog filters are cheap, fast, and have a large dynamic range in both amplitude and frequency. Digital filters, in comparison, are vastly superior in the level of performance that can be achieved. The entire transition occurs within only 1 hertz. It can't be expected from an op amp circuit. Digital filters can achieve thousands of times better performance than analog filters. This makes a dramatic difference in how filtering problems are approached. With analog filters, the emphasis is on handling limitations of the electronics, such as the accuracy and stability of the resistors and capacitors. In comparison, digital filters are so good that the performance of the filter is frequently ignored. The emphasis shifts to the limitations of the signals, and the theoretical issues regarding their processing.

It is common in DSP to say that a filter's input and output signals are in the time domain. This is because signals are usually created by sampling at regular intervals of time. But this is not the only way sampling can take place. The second most common way of sampling is at equal intervals in space. Many other domains are possible; however, time and space are by far the most common. When we see the term time domain in DSP, we remember that it may actually refer to samples taken over time, or it may be a general reference to any domain that the samples are taken in.

Every linear filter has an impulse response, a step response and a frequency response. Each of these responses contains complete information about the filter, but in a different form. If one of the three is specified, the other two are fixed and can be directly calculated. All three of these representations are important, because they describe how the filter will react under different circumstances. The most straightforward way to implement a digital filter is by convolving the input signal with the digital filter's impulse response. All possible linear filters can be made in this manner. When the impulse response is used in this way, filter designers give it a special name: the filter kernel.

There is also another way to make digital filters, called recursion. When a filter is implemented by convolution, each sample in the output is calculated by weighting the samples in the input, and adding them together. Recursive filters are an extension of this, using previously calculated values from the output, besides points from the input. Instead of using a filter kernel, recursive filters are defined by a set of recursion coefficients. For now, the important point is that all linear filters have an impulse response, even if we don't use it to implement the filter. The impulse responses of recursive filters are composed of sinusoids that exponentially decay in amplitude. In principle, this makes their impulse responses infinitely long. However, the amplitude eventually drops below the round-off noise of the system, and the remaining samples can be ignored. Because of this characteristic, recursive filters are also called Infinite Impulse Response or IIR filters. In comparison, filters carried out by convolution are called Finite Impulse Response or FIR filters.





Filter parameters. Every linear filter has an impulse response, a step response, and a frequency response. The step response, (b), can be found by discrete integration of the impulse response, (a). The frequency response can be found from the impulse response by using the Fast Fourier Transform (FFT), and can be displayed either on a linear scale, (c), or in decibels, (d).

As we know, the impulse response is the output of a system when the input is an impulse. In this same manner, the step response is the output when the input is a step (also called an edge, and an edge response). Since the step is the integral of the impulse, the step response is the integral of the impulse response. This provides two ways to find the step response: (1) feed a step waveform into the filter and see what comes out, or (2) integrate the impulse response. (To be mathematically correct: integration is used with continuous signals, while discrete integration, i.e., a running sum, is used with discrete signals). The frequency response can be found by taking the DFT (using the FFT algorithm) of the impulse response. The frequency response can be plotted on a linear vertical axis, such as in (c), or on a logarithmic scale (decibels), as shown in (d). The linear scale is best at showing the pass band ripple and roll-off, while the decibel scale is needed to show the stop band attenuation.

2.2 Representation of signals:

The most important part of any DSP task understands how information is contained in the signals we are working with. There are many ways that information can be contained in a signal. This is especially true if the signal is manmade. For instance, consider all of the modulation schemes that have been devised: AM, FM, single-sideband, pulse-code modulation, pulse-width modulation, etc. The list goes on and on. Fortunately, there are only two ways that are common for information to be represented in naturally occurring signals. We will call these: information represented in the time domain, and information represented in the frequency domain. Information represented in the time domain describes when something occurs and what the amplitude of the occurrence is. Each sample contains information that is interpretable without reference to any other sample. Even if we have only one sample from this signal, we still know something about what we are measuring. This is the simplest way for information to be contained in a signal.
In contrast, information represented in the frequency domain is more indirect. Many things in our universe show periodic motion. For example, a wine glass struck with a fingernail will vibrate, producing a ringing sound; the pendulum of a grandfather clock swings back and forth; stars and planets rotate on their axis and revolve around each other, and so forth. By measuring the frequency, phase, and amplitude of this periodic motion, information can often be obtained about the system producing the motion. Suppose we sample the sound produced by the ringing wine glass. The fundamental frequency and harmonics of the periodic vibration relate to the mass and elasticity of the material. A single sample, in itself, contains no information about the periodic motion,

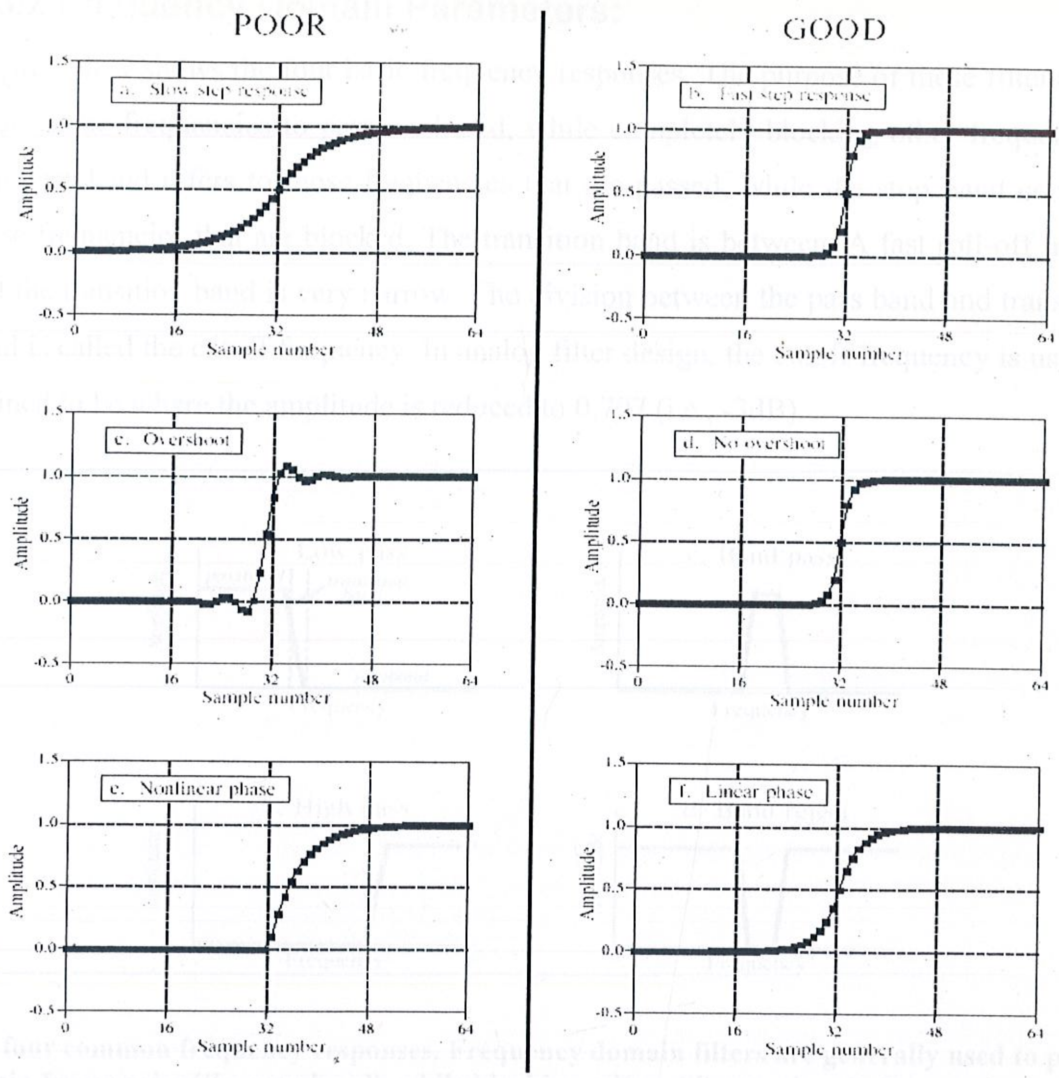
and therefore no information about the wine glass. The information is contained in the relationship between many points in the signal.

This brings us to the importance of the step and frequency responses. The step response describes how information represented in the time domain is being modified by the system. In contrast, the frequency response shows how information represented in the frequency domain is being changed. This distinction is absolutely critical in filter design because it is not possible to optimize a filter for both applications. Good performance in the time domain results in poor performance in the frequency domain, and vice versa.

2.3 Domain parameters

2.3.1 Time Domain Parameters:

It may not be obvious why the step response is of such concern in time domain filters. We may be wondering why the impulse response isn't the important parameter. The answer lies in the way that the human mind understands and processes information. Remember that the step, impulse and frequency responses all contain identical information, just in different arrangements. The step response is useful in time domain analysis because it matches the way humans view the information contained in the signals. Some of the regions may be smooth; others may have large amplitude peaks; others may be noisy. This segmentation is accomplished by identifying the points that separate the regions. This is where the step function comes in. The step function is the purest way of representing a division between two dissimilar regions. It can mark when an event starts, or when an event ends. It tells that whatever is on the left is somehow different from whatever is on the right. This is how the human mind views time domain information: a group of step functions dividing the information into regions of similar characteristics.
The step response, in turn, is important because it describes how the dividing lines are being modified by the filter. The step response parameters that are important in filter design are shown in Fig.. To distinguish events in a signal, the duration of the step response must be shorter than the spacing of the events. This dictates that the step response should be as fast (the DSP jargon) as possible. This is shown in Figs. (a) & (b). The most common way to specify the rise time is to quote the number of samples between the 10% and 90% amplitude levels. There are many reasons, noise reduction, inherent limitations of the data acquisition system, avoiding aliasing, etc.



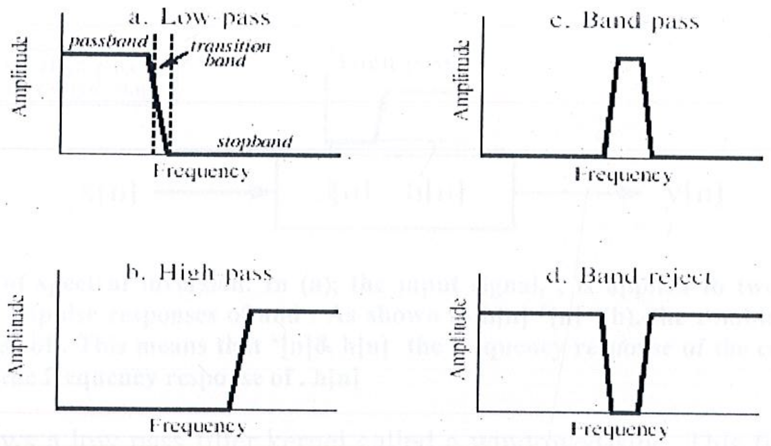
Parameters for evaluating time domain performance. The step response is used to measure how well a filter performs in the time domain. Three parameters are important: (1) transition speed (rise time), shown in (a) and (b), (2) overshoot, shown in (c) and (d), and (3) phase linearity (symmetry between the top and bottom halves of the step), shown in (e) and (f).

Figures (c) and (d) shows the next parameter that is important: overshoot in the step response. Overshoot must generally be eliminated because it changes the amplitude of samples in the signal; this is a basic distortion of the information contained in the time domain. This can be summed up in one question:

Finally, it is often desired that the upper half of the step response be symmetrical with the lower half, as illustrated in (e) and (f). This symmetry is needed to make the rising edges look the same as the falling edges. This symmetry is called linear phase, because the frequency response has a phase that is a straight line.

2.3.2 Frequency Domain Parameters:

Figure below shows the four basic frequency responses. The purpose of these filters is to allow some frequencies to pass unaltered, while completely blocking other frequencies. The pass band refers to those frequencies that are passed, while the stop band contains those frequencies that are blocked. The transition band is between. A fast roll-off means that the transition band is very narrow. The division between the pass band and transition band is called the cutoff frequency. In analog filter design, the cutoff frequency is usually defined to be where the amplitude is reduced to 0.707 (i.e., -3dB).

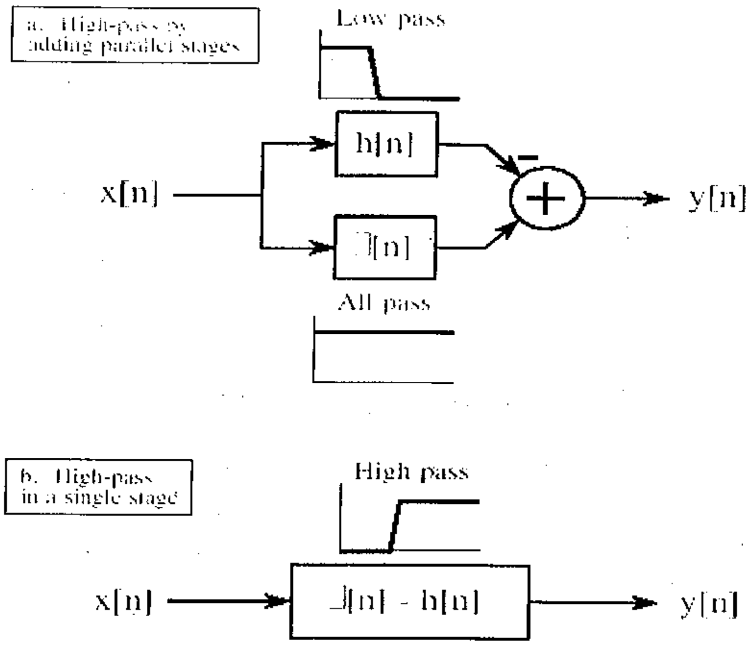


The four common frequency responses. Frequency domain filters are generally used to pass certain frequencies (the pass band), while blocking others (the stop band). Four responses are the most common: low-pass, high-pass, band-pass, and band-reject.

Digital filters are less standardized, and it is common to see 99%, 90%, 70.7%, and 50% amplitude levels defined to be the cutoff frequency. Figure shows three parameters that measure how well a filter performs in the frequency domain. To separate closely spaced frequencies, the filter must have a fast roll-off, as illustrated in (a) and (b). For the pass band frequencies to move through the filter unaltered, there must be no pass band ripple, as shown in (c) and (d). Lastly, to adequately block the stop band frequencies, it is necessary to have good stop band attenuation, displayed in (e) and (f).

2.4 High-Pass, Band-Pass and Band-Reject Filters

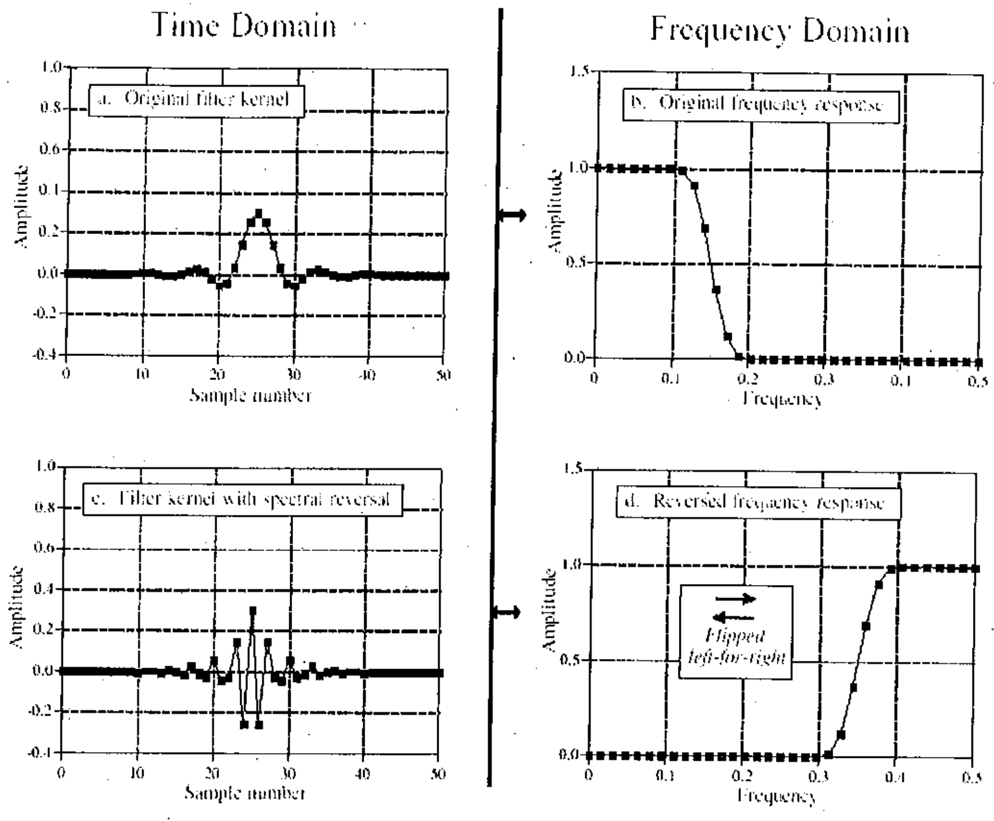
High-pass, band-pass and band-reject filters are designed by starting with a low-pass filter, and then converting it into the desired response. For this reason, most discussions on filter design only give examples of low-pass filters. There are two methods for the low-pass to high-pass conversion: spectral inversion and spectral reversal.



a. High-pass by adding parallel stages
Low pass
h[n]
x[n]
y[n]
δ[n]
All pass
b. High-pass in a single stage
High pass
x[n]
δ[n] - h[n]
y[n]

Block diagram of spectral inversion. In (a), the input signal, , is applied to two x[n] systems in parallel, having impulse responses of and . As shown in h[n] *[n] (b), the combined system has an impulse response of . This means that *[n]& h[n] the frequency response of the combined system is the inversion of the frequency response of . h[n]

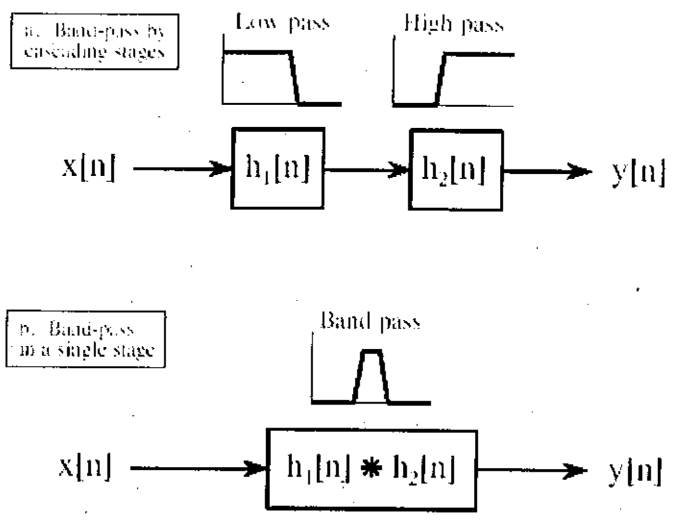
Figure (a) shows a low pass filter kernel called a windowed-sinc. This filter kernel is 51 points in length, although many of samples have a value so small that they appear to be zero in this graph. The corresponding frequency response is shown in (b), found by adding 13 zeros to the filter kernel and taking a 64 point FFT. Two things must be done to change the low-pass filter kernel into a high-pass filter kernel. First, change the sign of each sample in the filter kernel. Second, add one to the sample at the center of symmetry. This results in the high-pass filter kernel shown in (c), with the frequency response shown in (d). Spectral inversion flips the frequency response top-for-bottom, changing the pass bands into stop bands, and the stop bands into pass bands. In other words, it changes a filter from low-pass to high-pass, high-pass to low-pass, band-pass to band-reject, or band-reject to band-pass.



Example of spectral reversal . The low-pass filter kernel in (a) has the frequency response shown in (b). A high-pass filter kernel, (c), is formed by changing the sign of every other sample in (a). This action in the time domain results in the frequency domain being flipped left-for-right, resulting in the high-pass frequency response shown in (d)

Figure shows why this two step modification to the time domain results in an inverted frequency spectrum. In (a), the input signal, , is applied to x[n] two systems in parallel. One of these systems is a low-pass filter, with an impulse response given by . The other system does nothing to the signal, h[n] and therefore has an impulse response that is a delta function, . The *[n] overall output, is equal to the output of the all-pass system minus the y[n] output of the low-pass system. Since the low frequency components are subtracted from the original signal, only the high frequency components appear in the output. Thus, a high-pass filter is formed. This could be performed as a two step operation in a computer program: run the signal through a low-pass filter, and then subtract the filtered signal from the original. However, the entire operation can be performed in a signal stage by combining the two filter kernels. As described parallel systems with added outputs can be combined into a single stage by adding their impulse responses. As shown in (b), the filter kernel for the high pass filter is given by: That is, change the sign of all the samples, *[n] & h[n] and then add one to the sample at the center of symmetry.

The second method for low-pass to high-pass conversion, spectral reversal, is illustrated in Fig. Just as before, the low-pass filter kernel in (a) corresponds to the frequency response in (b). The high-pass filter kernel, (c), is formed by changing the sign of every other sample in (a). As shown in (d), this flips the frequency domain left-for-right: 0 becomes 0.5 and 0.5 becomes 0. The cutoff frequency of the example low-pass filter is 0.15, resulting in the cutoff frequency of the high-pass filter being 0.35



Designing a band-pass filter. As shown in (a), a band-pass filter can be formed by cascading a low-pass filter and a high-pass filter. This can be reduced to a single stage, shown in (b). The filter kernel of the single stage is equal to the convolution of the low-pass and high pass filter kernels.

Changing the sign of every other sample is equivalent to multiplying the filter kernel by a sinusoid with a frequency of 0.5. As discussed in Chapter 10, this has the effect of shifting the frequency domain by 0.5. Look at (b) and imagine the negative frequencies between -0.5 and 0 that are of mirror image of the frequencies between 0 and 0.5. The frequencies that appear in (d) are the negative frequencies from (b) shifted by 0.5. Lastly, above figure show how low-pass and high-pass filter kernels can be combined to form band-pass and band-reject filters. In short, adding the filter kernels produces a band-reject filter, while convolving the filter kernels produces a band-pass filter. These are based on the way cascaded and parallel systems are be combined. Multiple combinations of these techniques can also be used. For instance, a band-pass filter can be designed by adding the two filter kernels to form a stop-pass filter, and then use spectral inversion or spectral reversal as previously described. All these techniques work very well with few surprises.

CHAPTER-3

DESIGNING FILTER

3.1 How to design filters

3.1.1 Different classes and equations:

Two important classes of filters based on impulse response type:

Finite Impulse Response (FIR)

Infinite Impulse Response (IIR)

Expressing filter functions:

System function representation;

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 + \sum_{k=1}^{N} a_k z^{-k}} \qquad (1)$$

Difference Equation representation:-

$$\sum_{k=0}^{N} a_k y(n-k) = \sum_{k=0}^{M} b_k x(n-k) \qquad (2)$$

- Each of this form allows various methods of implementations.
- The eq (2) can be viewed as a computational procedure (an algorithm) for determining the output sequence y(n) of the system from the input sequence x(n)
- Different realizations possible with different arrangements of eq (2)

3.1.2 Filter Design Issues:

- Realizable
- Stable
- Sharp Cutoff Characteristics
- Minimum order
- Generalized procedure
- Linear phase characteristics

Issues considered for filter implementation:

Simple design

Structured ness – modularity

Generalization of design – any filter type

Cost of implementation

Software/hardware realization

IIR:

- Out put is a function of past o/p, present and past i/p's
- Recursive nature
- Poles and zeros
- Sharp cutoff achievable with min order
- Difficult to have linear phase over full range of freq.
- Design procedure; analog design – conversion from analog to digital

FIR

- Inherently Stable
- Linear phase characteristics possible
- Simple implementation – both recursive and non recursive structures possible
- Free of limit cycle oscillations when implemented on a finite-word length digital system

Disadvantages:

- Sharp cutoff at the cost of higher order
- Higher order leading to more delay, more memory and higher cost of implementation

Importance of Linear Phase:

The group delay is defined as

$$\tau_g = -\frac{d\theta(\omega)}{d\omega}$$

- Nonlinear phase results in different frequencies experiencing different delay and arriving at different time at the receiver
- This creates problems with speech processing and data communication applications

3.1.3 Understanding simple filtering operation:

1. Unity Gain Filter

   y(n)=x(n)

2. Constant gain filter

   y(n)=Kx(n)

3. Unit delay filter

   y(n)=x(n-1)

4. Two - term Difference filter

   y(n) = x(n)-x(n-1)

5. Two-term average filter

   y(n) = 0.5(x(n)+x(n-1))

6. Three-term average filter (3-point moving average filter)

   y(n) = 1/3[x(n)+x(n-1)+x(n-2)]

7. Central Difference filter

   y(n)= 1/2[x(n) – x(n-2)]

- Order of the filter is the number of previous inputs used to compute the current output
- Filter coefficients are the numbers associated with each of the terms x(n), x(n-1),.. Etc

Ex.	order	a0	a1	a2
1	0	1	-	-
2	0	K	-	-
3	1	0	1	-
4(HP)	1	1	-1	-
5(LP)	1	1/2	1/2	-
6(LP)	2	1/3	1/3	1/3
7(HP)	2	1/2	0	-1/2

3.1.4 Design of FIR filters:

Symmetric and Antisymmetric FIR filters:

- Symmetry in filter impulse response will ensure Linear phase

Comments on filter coefficients:

- The number of filter coefficients that specify the frequency response in (M+1)/2 when is M odd and M/2 when M is even in case of symmetric conditions
- In case of impulse response antisymmetric h(M-1/2)=0 so that there are (M-1/2) filter coefficients when M is odd and M/2 coefficients when M is even

Choice of Symmetric and antisymmetric unit sample response:-

- If h(n)=-h(M-1-n) and M is odd, Hr(w) implies that Hr(0)=0 & Hr(π)=0, consequently not suited for low pass and high pass filter.
- Similarly if M is even Hr(0)=0 hence not used for low pass filter
- Hence antisymetric condition is not generally used
- Symmetry condition h(n)=h(M-1-n) yields a linear-phase FIR filter with non zero response at w=0 if desired.

Zeros of Linear Phase FIR Filters:

Consider the filter system function

$$H(z) = \sum_{n=o}^{M-1} h(n)z^{-n}$$

Expanding this equation:

$$H(z) = h(0) + h(1)z^{-1} + h(2)z^{-2} + \ldots\ldots + h(M-2)z^{-(M-2)} + h(M-1)z^{-(M-1)}$$

$$\text{since } for\ Linear-phase we\ need$$

$$h(n) = h(M-1-n) \quad i.e.,$$

$$h(0) = h(M-1); h(1) = h(M-2); \ldots\ldots h(M-1) = h(0);$$

$$then$$

$$H(z) = h(M-1) + h(M-2)z^{-1} + \ldots\ldots\ldots + h(1)z^{-(M-2)} + h(0)z^{-(M-1)}$$

3.2 Different methods

3.2.1 Methods of designing FIR filters:

1. Fourier series based method
2. Window based method
3. Frequency sampling method

3.2.2 FIR Filter Design by the Window Method:

The applet uses the window method, a widely-used approach to FIR filter design. The theoretical basis of the method is simple and elegant. If the discrete Fourier transform (DFT) is applied to a function representing the required filter frequency response, we obtain the impulse response of the required filter. Since the sampled impulse response values for an FIR filter correspond to the sequence of filter coefficients, we therefore obtain the required filter coefficients directly.

Unfortunately, the filter obtained by this technique suffers from two main drawbacks:

1. The filter is non-causal.
2. The order of the filter is infinitely large.

A "non-causal" filter is one which requires not only the current and previous input values (x_n, x_{n-1}, x_{n-2}, x_{n-3}, ...) but also future input values (x_{n+1}, x_{n+2}, x_{n+3}, ...). Non-causal filters can be used where a complete record of the sampled input signal is available, but if the filter is operating in real time, future input values are unknown.

To make the filter causal, so that it uses only the current and previous input values, the filter coefficients must be "time-shifted". This effectively means delaying the input samples by a sufficient number of sampling intervals; as a result, the filter output is delayed by the same length of time. In most cases, this is not a serious problem.

The fact that the order of the filter is infinite is, however, rather more of a difficulty. An infinite order means that an infinite number of terms must be calculated to obtain the filter output; clearly this is not practical. If the filter is to operate in real time, the maximum order is usually limited by the need for the processor to perform the filtering calculations within a specified time to allow it to keep up with the signal data coming in every sampling interval.

To restrict the order of the filter to a finite value, we must again make some adjustments to the filter coefficients. The technique used is to multiply the sequence of filter coefficients by a tapered window function of a finite width. The simplest window function is the rectangular window. Multiplying the sequence of filter coefficients by a rectangular window corresponds to truncating (chopping off) the sequence after a certain number of terms.

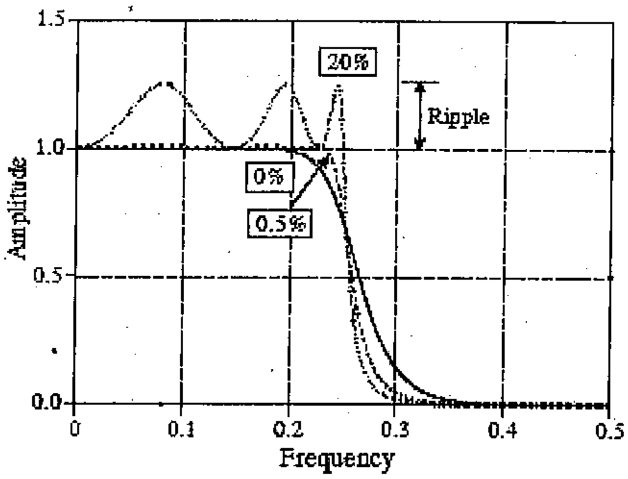
The drawback of truncating the filter coefficients to produce a "cut-down" finite-order version of the filter is that it no longer has the ideal frequency response originally used as the basis for designing the filter. Instead of a perfectly flat pass band in the gain frequency response curve, ripples tend to appear; instead of an infinitely sharp cutoff at the design cutoff frequency, there is a more gradual transition between the pass band and stop band. In the stop band itself, the filter gain is no longer exactly zero, but has small nonzero values following an oscillating curve, with peaks in the gain at certain points in the stop band. To make the filter frequency response approximate more closely to the ideal, the width of the window must be increased to include more coefficients, i.e. the order of the filter has to be increased.

Various window functions can be used in the design of FIR filters. Each window function has its good and bad points. As indicated earlier, the choice of window function determines the minimum amount of attenuation provided in the stop band; the rectangular window, for example, has rather poor stop band performance, while the Blackman window gives much better stop band attenuation. The other side of the coin is that the Blackman filter has rather poor transition band performance, while the rectangular window gives a sharp cutoff (narrow transition band). To achieve a given transition band performance with a Blackman window, for example, a much higher filter order is needed than would be the case with the rectangular window.

3.3 Chebyshev Filters

The Chebyshev response is a mathematical strategy for achieving a faster roll-off by allowing ripple in the frequency response. Analog and digital filters that use this approach are called Chebyshev filters. For instance, analog Chebyshev filters were used for analog-to-digital and digital-to-analog conversion. These filters are named from their use of the Chebyshev polynomials, developed by the Russian mathematician Pafnuti Chebyshev (1821-1894). This name has been translated from Russian and appears in the literature with different spellings, such as: Chebychev, Tschebyscheff, Tchebysheff and Tchebichef.

Figure shows the frequency response of low-pass Chebyshev filters with passband ripples of: 0%, 0.5% and 20%. As the ripple increases (bad), the roll-off becomes sharper (good). The Chebyshev response is an optimal trade-off between these two parameters. When the ripple is set to 0%, the filter is called a maximally flat or Butterworth filter (after S. Butterworth, a British engineer who described this response in 1930). A ripple of 0.5% is a often good choice for digital filters. This matches the typical precision and accuracy of the analog electronics that the signal has passed through. The Chebyshev filters discussed are called type 1 filters, meaning that the ripple is only allowed in the pass band.



The Chebyshev response . Chebyshev filters achieve a faster roll-off by allowing ripple in the Pass band. When the ripple is set to 0%, it is called a maximally flat or Butterworth filter. Consider using a ripple of 0.5% in your designs; this pass band unflatness is so small that it Can not be seen in this graph, but the roll-off is much faster than the Butterworth.

In comparison, type 2 Chebyshev filters have ripple only in the stop band. Type 2 filters are seldom used, and we won't discuss them. There is, however, an important
Design called the elliptic filter, which has ripple in both the pass band and the stop band. Elliptic filters provide the fastest roll-off for a given number of poles, but are much harder to design. We won't discuss the elliptic filter here, but be aware that it is frequently the first choice of professional filter designers, both in analog electronics and DSP. If you need this level of performance, buy a software package for designing digital filters.

CHAPTER-4

Digital Signal Processing

4.1 Why DSP:

The issues introduced in this article include:

- DSP overview
- Real-time DSP operation
- Real-world signals
- Sampling rates and anti-alias filtering
- DSP algorithm
- Modeling filter transform functions
- Relating the models to DSP architecture
- Experimenting with digital filters

Having heard a lot about digital signal processing (DSP) technology, we may have wanted to find out what can be done with DSP. This series is an introduction to DSP topics from the point of view of analog system designers seeking additional tools for handling analog signals. Designers reading this series can learn about the possibilities of DSP to deal with analog signals and where to find additional sources of information and assistance.

4.1.1 What is DSP?

In brief, DSPs are processors or microcomputers whose hardware, software, and instruction sets are optimized fot high-speed numeric processing applications an essential for processing digital data representing analog signals in real time. What a DSP does is straightforward. When acting as a digital filter, for example, the DSP receives digital values based on samples of a signal, calculates the results of a filter function operating on these values, and provides digital values that represent the filter output; it can also provide system control signals based on properties of these values. The DSP's high-speed arithmetic and logical hardware is programmed to rapidly execute algorithms modeling the filter transformation.

The combination of design elements arithmetic operators, memory handling, instruction set, parallelism, data addressing that provide this ability forms the key difference between DSPs and other kinds of processors. Understanding the relationship between real-time signals and DSP calculation speed provides some background on just how special this combination is. The real-time signal comes to the DSP as a train of individual samples from an analog-to-digital converter (ADC). To do filtering in real-time, the DSP must complete all the calculations and operations required for processing each sample (usually updating a process involving many previous samples) before the next sample arrives. To perform high-order filtering of real-world signals having significant frequency content calls for really fast processors.

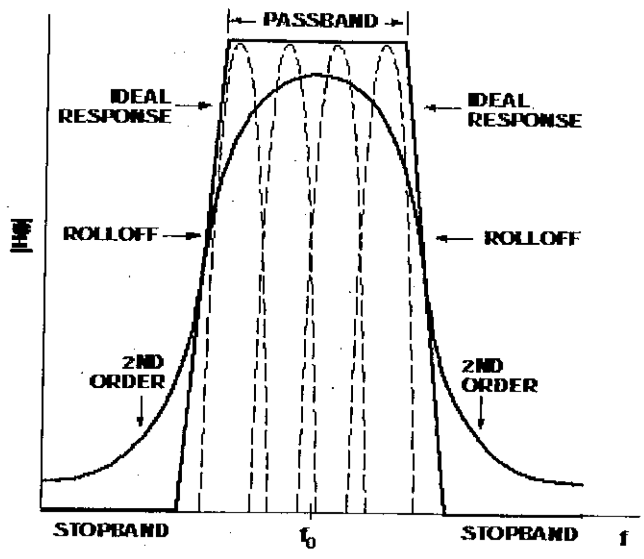
4.1.2 WHY USE A DSP?

To get an idea of the type of calculations a DSP does and get an idea of how an analog circuit compares with a DSP system, one could compare the two systems in terms of a filter function. The familiar analog filter uses resistors, capacitors, inductors, amplifiers. It can be cheap and easy to assemble, but difficult to calibrate, modify, and maintain a difficulty that increases exponentially with filter order. For many purposes, one can more easily design, modify, and depend on filters using a DSP because the filter function on the DSP is software-based, flexible, and repeatable. Further, to create flexibly adjustable filters with higher-order response requires only software modifications, with no additional hardware unlike purely analog circuits. An ideal band pass filter, with the frequency response shown in Figure 1 below, would have the following characteristics:

- a response within the pass band that is completely flat with zero phase shift
- Infinite attenuation in the stop band.

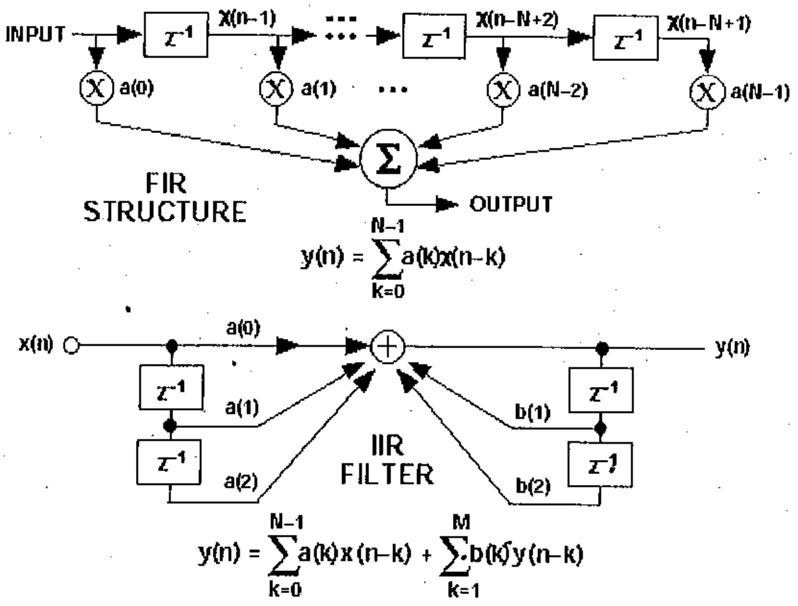
  Useful additions would include:
- Pass band tuning and width control
- Stop band roll off control.

As Figure shows, an analog approach using second-order filters would require quite a few staggered high-Q sections; the difficulty of tuning and adjusting it can be imagined.



An ideal bandpass filter and second-order approximations

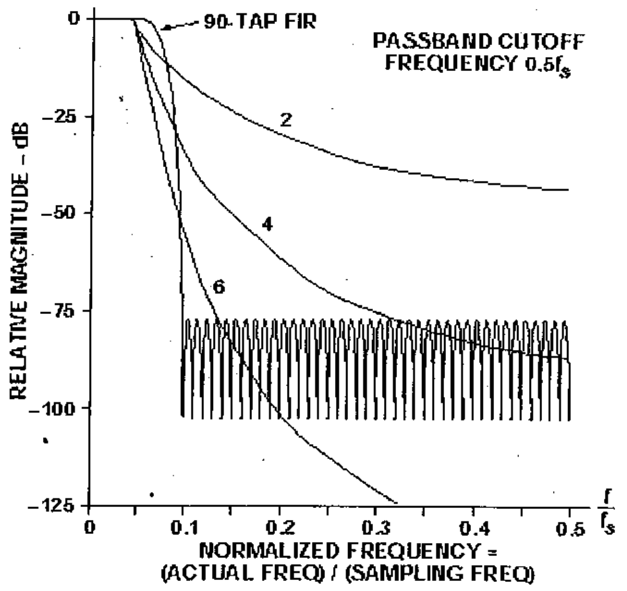
With DSP software, there are two basic approaches to filter design: finite impulse response (FIR) and infinite impulse response (IIR). The FIR filter's time response to an impulse is the straightforward weighted sum of the present and a finite number of previous input samples. Having no feedback, its response to a given sample ends when the sample reaches the "end of the line" below. An FIR filter's frequency response has no poles, only zeros. The IIR filter, by comparison, is called infinite because it is a recursive function: its output is a weighted sum of inputs and outputs. Since it is recursive, its response can continue indefinitely. An IIR filter frequency response has both poles and zeros.



Filter equations and delay-line representation.

The xs are the input samples, ys are the output samples, as are input sample weightings, and bs are output sample weightings. n is the present sample time, and M and N are the number of samples programmed (the filter's order). Note that the arithmetic operations indicated for both types are simply sums and products in potentially great number. In fact, multiply-and-add is the case for many DSP algorithms that represent mathematical operations of great sophistication and complexity.

Approximating an ideal filter consists of applying a transfer function with appropriate coefficients and a high enough order, or number of taps (considering the train of input samples as a tapped delay line). Below shows the response of a 90-tap FIR filter compared with sharp-cutoff Chebyshev filters of various orders. The 90-tap example suggests how close the filter can come to approximating an ideal filter. By comparison, it would not be cost-effective to attempt this level of approximation with a purely analog circuit. Another crucial point in favor of using a DSP to approximate the ideal filter is long-term stability. With an FIR (or an IIR having sufficient resolution to avoid truncation-error buildup), the programmable DSP achieves the same response, time after time. Purely analog filter responses of high order are less stable with time.



90-tap FIR filter response compared with those of sharp cutoff Chebyshev filters.

Mathematical transform theory and practice are the core requirement for creating DSP applications and understanding their limits. This article series walks through a few signal-analysis and -processing examples to introduce DSP concepts. The series also provides references to texts for further study and identifies software tools that ease the development of signal-processing software.

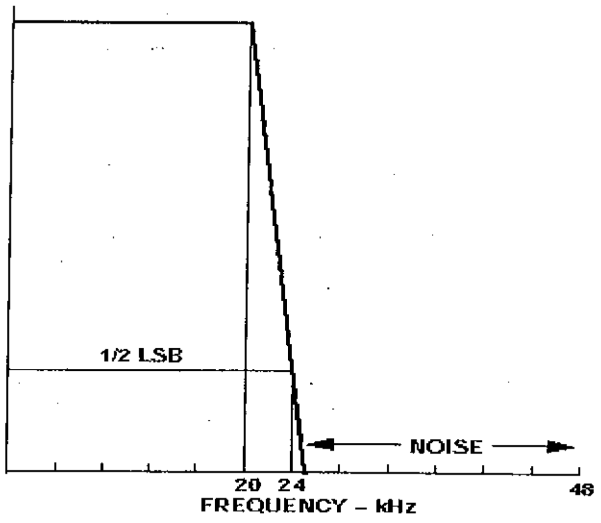
4.2 Real-time DSP operation

4.2.1 SAMPLING REAL WORLD PROBLEMS:

Real-world phenomena are analog the continuously changing energy levels of physical processes like sound, light, heat, electricity, magnetism. A transducer converts these levels into manageable electrical voltage and current signals, and an ADC samples and converts these signals to digital for processing. The conversion rate, or sampling frequency, of the ADC is critically important in digital processing of real-world signals.

This sampling rate is determined by the amount of signal information that is needed for processing the signals adequately for a given application. In order for an ADC to provide enough samples to accurately describe the real-world signal, the sampling rate must be at least twice the highest-frequency component of the analog signal. For example, to accurately describe an audio signal containing frequencies up to 20 kHz, the ADC must sample the signal at a minimum of 40 kHz. Since arriving signals can easily contain component frequencies above 20 kHz (including noise), they must be removed before sampling by feeding the signal through a low-pass filter ahead of the ADC. This filter, known as an anti-aliasing filter, is intended to remove the frequencies above 20 kHz that could corrupt the converted signal.

However, the anti-aliasing filter has a finite frequency roll off, so additional bandwidth must be provided for the filter's transition band. For example, with an input signal bandwidth of 20 kHz, one might allow 2 to 4 kHz of extra bandwidth.

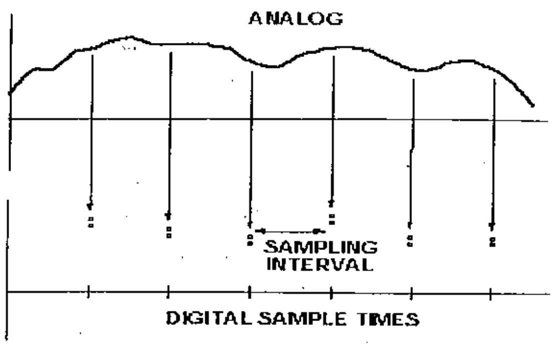


Antialiasing filter ideal response.

Above depicts the filter needed to reject any signals with frequencies above half of a 48-kHz sampling rate. Rejection means attenuation to less than 1/2 least-significant bit (LSB) of the ADC's resolution. One way to achieve this level of rejection without a highly sophisticated analog filter is to use an over sampling converter, such as a sigma-delta ADC. It typically obtains low-resolution (e.g., 1-bit) samples at megahertz rates much faster than twice the highest frequency component greatly easing the requirement for the analog filter ahead of the converter. An internal digital filter restores the required resolution and frequency response.

4.2.2 PROCESSING REAL-WORLD SIGNALS:

The ADC sampling rate depends on the bandwidth of the analog signal being sampled. This sampling rate sets the pace at which samples are available for processing. Once the system bandwidth requirements have established the A/D converter sampling rate, the designer can begin to explore the speed requirements of the DSP processor. Processing speed at a required sample rate is influenced by algorithm complexity. As a rule, the DSP needs to finish all operations relating to the first sample before receiving the second sample. The time between samples is the time budget for the DSP to perform all processing tasks. For the audio example, a 48-kHz sampling rate corresponds to a 20.833-μs sampling interval. Figure 5 relates the analog signal and digital sampling rate.



Sampling train and processing time

Next consider the relation between the speed of the DSP and complexity of the algorithm (the software containing the transform or other set of numeric operations). Complex algorithms require more processing tasks. Because the time between samples is fixed, the higher complexity calls for faster processing. For example, suppose that the algorithm requires 50 processing operations to be performed between samples. Using the previous example's 48-kHz sampling rate (20.833-μs sampling interval), one can calculate the minimum required DSP processor speed, in millions of operations per second (MOPS) as follows

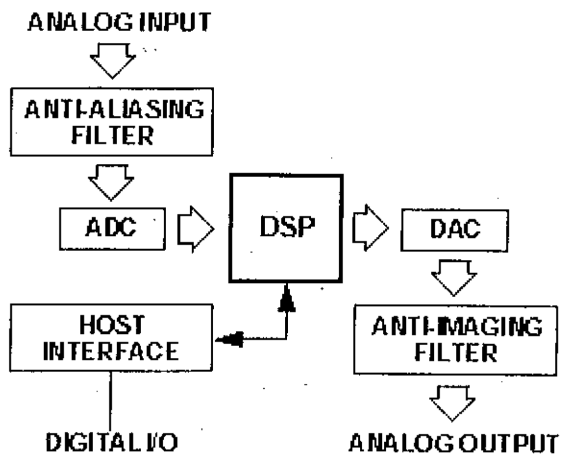
$$DSP\ Speed = \frac{Operations}{Sampling\ Interval} = \frac{50}{20.833\ \mu s} = 2.4\ \text{MOPS}$$

Thus if all of the time between samples is available for operations to implement the algorithm, a processor with a performance level of 2.4 MOPS is required. Note that the two common ratings for DSPs, based on operations per second (MOPS) and instructions per second (MIPS), are not the same. A processor with a 10-MIPS rating that can perform 8 operations per instruction has basically the same performance as a faster processor with a 40 MIPS rating that can only perform 2 operations per instruction.

4.3 DSP algorithm

4.3.1 DEVELOPING A DSP SYSTEM:

Having discussed the role of the processor, the ADC, the anti-aliasing filter, and the timing relationships between these components, it is time to look at a complete DSP system. Diagram below shows the building blocks of a typical DSP system that could be used for data acquisition and control



Putting together elements of a DSP system.

Few components make up the DSP system, because so much of the system's functionality comes from the programmable DSP. Converters funnel data into and out of the DSP; the ADC timing is controlled by a precise sampling clock. To simplify system design, many converter devices available today combine some or all of the following: an A/D converter, a D/A converter, a sampling clock, and filters for anti-aliasing and anti-imaging. The clock oscillator in these types of I/O components is separately controlled by an external crystal. Here are some important points about the data flow in this sort of DSP system:

Analog Input: The analog signal is appropriately band-limited by the anti-aliasing filter and applied to the input of the ADC. At the selected sampling time, the converter interrupts the DSP processor and makes the digital sample available. The choice between serial and parallel interfacing between the ADC and DSP depends on the amount of data, design complexity trade-offs, space, power, and price.

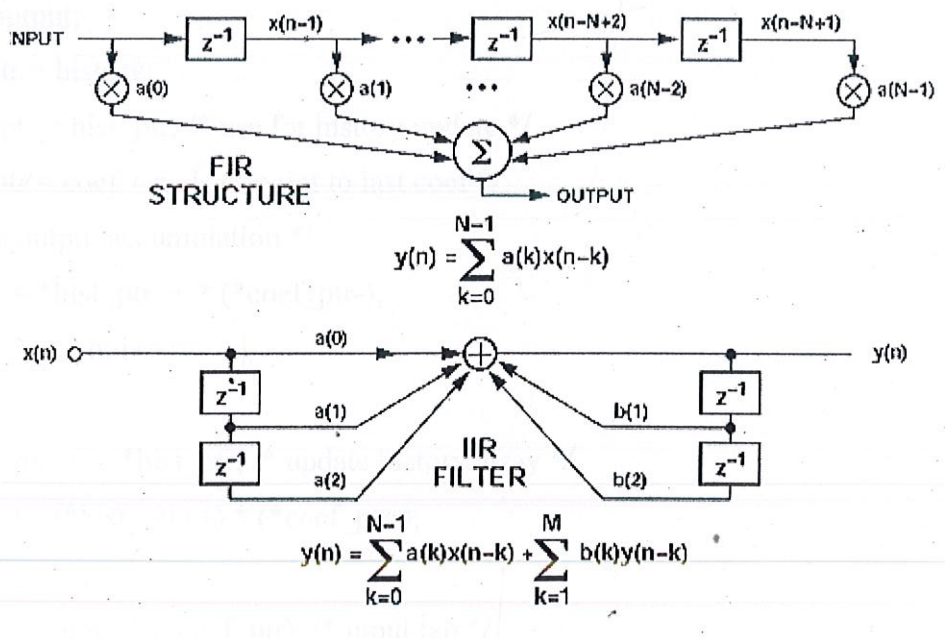
Digital Signal Processing: The incoming data is handled by the DSP's algorithm software. When the processor completes the required calculations, it sends the result to the DAC. Because the signal processing is programmable, considerable flexibility is available in handling the data and improving system performance with incremental programming adjustments.

Analog Output: The DAC converts the DSP's output into the desired analog output at the next sample clock. The converter's output is smoothed by a low-pass, anti-imaging filter (also called a reconstruction filter), to produce the reconstructed analog signal.

Host Interface: An optional host interface lets the DSP communicate with external systems, sending and receiving data and control information.

4.4 Modeling Filter Transform Functions:

The three principal reasons for using digital filtering are (1) closer approach to ideal filter approximations, (2) ability to adjust filter characteristics in software rather than by physical tuning, and (3) compatibility of filter response with sampled data. The two best-known filters are the finite impulse-response (FIR) and infinite impulse-response (IIR) types. The FIR filter response is called finite because its output is based solely on a finite set of input samples; it is non-recursive and has no poles, only zeroes in its s-plane. The IIR filter, on the other hand, has a response that can go on indefinitely (and can be unstable) because it is recursive, i.e., its output values are affected by both input and output. It has both poles and zeroes in its s-plane. Diagram below shows the typical filter architectures and summation formulas that appeared previously.



Filter equations and their delay-line models.

To model these filters digitally, one might take two steps. First, view these formulas as programs running on a computer. This step consists of breaking down the formula into the mathematical steps (e.g., multiply and add) and identifying all of the additional operations that would be necessary for a computer to perform (handling instructions and data, testing status, etc.) to implement the formula in software.

Second, take those operations and write them as a program. This can be a fairly arduous task. Fortunately, there is much "canned" software available, often in a high-level language (HLL) such as C, somewhat simplifying (but by no means eliminating!) the job of programming. From the point of view of learning, though, it may be more instructive to start with assembly language; also assembly language algorithms are often more useful than HLL where system performance must be optimized. At the level of abstraction of some high-level languages, the program may not look much like the equations. For example, below shows an example of an FIR algorithm implemented as a C program.

```
float fir_filter(float input, float *coef, int n, float *history)
{
int i;
float *hist_ptr, *hist1_ptr, *coef_ptr;
float output;
hist_ptr = history;
hist1_ptr = hist_ptr; /* use for history update */
coef_ptr = coef + n -1; /* point to last coef */
/*form output accumulation */
output = *hist_ptr++ * (*coef_ptr-);
for(i = 2; i < n; i++)
{
*hist1_ptr++ = *hist_ptr; /* update history array */
output += (*hist_ptr++) * (*coef_ptr-);
}
output += input * (*coef_ptr); /* input tap */
*hist1_ptr = input; /* last history */
return(output);
}
```

FIR Filter as C program:

There are many analysis packages available that support algorithm modeling; see the references at the end of this article for several popular packages. Now, continuing the discussion of the process, after these filter algorithms have been modeled, they are ready for implementation in DSP architecture.

CHAPTER-5

Implementing Filters on ADSP 2181

5.1 Why ADSP 2181:

The ADSP-2100 Family Development Software, a complete set of tools for software and hardware system development, supports the ADSP-2181. The System Builder provides a high level method for defining the architecture of systems under development. The Assembler has an algebraic syntax that is easy to program and debug. The Linker combines object files into an executable file. The Simulator provides an interactive instruction-level simulation with a reconfigurable user interface to display different portions of the hardware environment. The C Compiler, based on the Free Software Foundation's C Compiler, generates ADSP-2181 assembly source code. The source code debugger allows programs to be corrected in the C environment. The Runtime Library includes over 100 ANSI-standard mathematical and DSP-specific functions. The EZ-KIT Lite is a hardware/software kit offering a complete development environment for the entire ADSP-21xx family:an ADSP-2181 evaluation board with PC monitor software plus Assembler, Linker, Simulator, and PROM Splitter software. The ADSP-218x EZ-KIT Lite is a low-cost, easy to use hardware platform on which you can quickly get started with your DSP software design. The EZ-KIT Lite includes the following features

- 33 MHz ADSP-2181
- Full 16-bit Stereo Audio I/O with AD1847 Sound Port Codec
- RS-232 Interface to PC with Windows 3.1 Control Software
- Stand-Alone Operation with Socket EPROM
- EZ-ICE Connector for Emulator Control

The ADSP-218x EZ-ICE Emulator aids in the hardware debugging of ADSP-218x systems. The emulator consists of hardware, host computer resident software and the target board connector. The ADSP-218x integrates on-chip emulation support with a 14-

pin ICE-Port interface. This interface provides a simpler target board connection requiring fewer mechanical clearance considerations than other ADSP-2100 Family EZ-ICEs. The ADSP-218x device need not be removed from the target system when using the EZ-ICE, nor are any adapters needed. Due to the small footprint of the EZ-ICE connector, emulation can be supported in final board designs.

Apart from this we have blackfin processors with ADBF 535 which is also more ever like 2181 but for our use it is more compatible and handy at different scenes in our project.

5.1.1 ADSP 2181 with Visual DSP++:

The ADSP 2181 is compatible with visual dsp++3.5 software. In this software we can write programs in C,C++ and in assembly language. We have generally used assembly language because it gives less errors .

Visual DSP++ provides the following features.

- Extensive editing capabilities.
- Flexible project management.
- Easy access to code development tools.
- Flexible project build options.
- Visual DSP++ Kernel (VDK)
- Flexible workspace management..
- Easy movement between debug and build activities.

Visual DSP++ reduces your debugging time by providing these key features.

- Easy-to-use debugging activities.
- Multiple language support
- Effective debug control.
- Tools for improving performance.

5.1.2 Code Development Tools:

Code development tools for 16-bit processors include:

- C/C++ compiler
- Runtime library with over 100 math, DSP, and C runtime library routines

- Assembler
- Linker
- Splitter
- Loader
- Simulator
- Emulator (must be purchased separately from Visual DSP++)

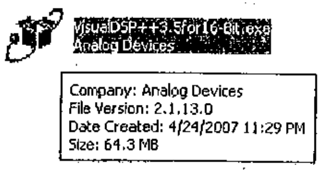
These tools enable you to develop applications that take full advantage of our processor's architecture. The Visual DSP++ linker supports multiprocessing, shared memory, and memory overlays. The code development tools provide the following key features.

- **Easy-to-program C, C++, and assembly languages**. Program in C/C++, assembly, or mix C/C++ and assembly in one source. The assembly language is based on an algebraic syntax that is easy to learn, program, and debug.
- **Flexible system definition.** Define multiple types of executables for a single type of processor in one Linker Description File (.LDF). Specify input files, including objects, libraries, shared memory files, overlay files, and executables.
- **Support for overlays, multiprocessors, and shared memory executables**. The linker places code and resolves symbols in multiprocessor memory space for use by multiprocessor systems. The loader enables you to configure multiprocessors with less code and faster boot time. Create host, link port, and PROM boot images.

5.2 Installing the software and interfacing it with ADSP 2181

First of all we get the software from website; i.e., analog.com. The software is valid for 90 days only, it's a trial version. Now point is how to install it and interface it with the ADSP2181 kit.

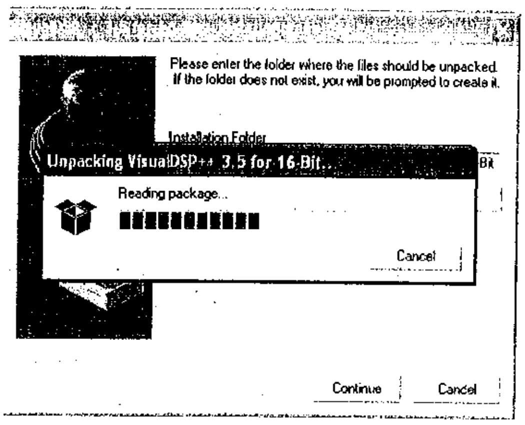
5.2.1 Installing the software:


VisualDSP++3.5for16-Bit.exe
Analog Devices

Company: Analog Devices
File Version: 2.1.13.0
Date Created: 4/24/2007 11:29 PM
Size: 64.3 MB

(software)

As we have earlier stated that we have got the software from analog.com which is of 64 mb in size. Now following are the software installation steps.

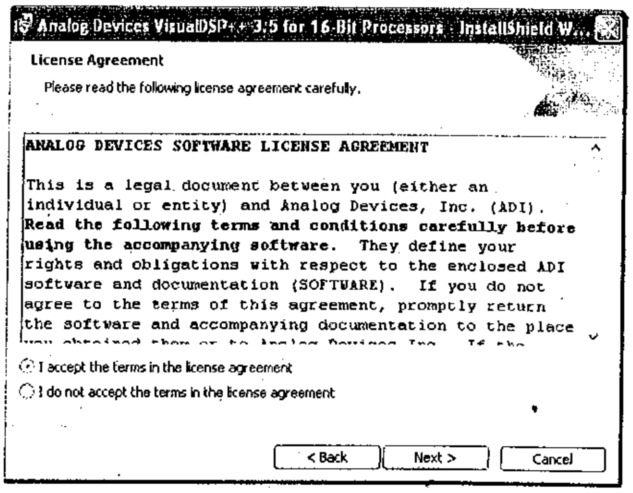

Please enter the folder where the files should be unpacked.
If the folder does not exist, you will be prompted to create it.

Installation Folder

Unpacking VisualDSP++ 3.5 for 16-Bit..

Reading package...

Step-1(Installing the software)


Analog Devices VisualDSP++ 3.5 for 16-Bit Processors - InstallShield W...

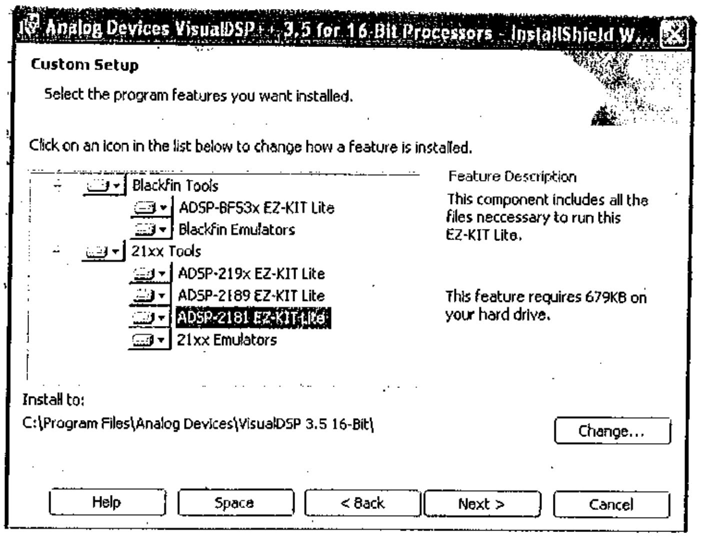
License Agreement

Please read the following license agreement carefully.

```
ANALOG DEVICES SOFTWARE LICENSE AGREEMENT

This is a legal document between you (either an
individual or entity) and Analog Devices, Inc. (ADI).
Read the following terms and conditions carefully before
using the accompanying software.  They define your
rights and obligations with respect to the enclosed ADI
software and documentation (SOFTWARE).  If you do not
agree to the terms of this agreement, promptly return
the software and accompanying documentation to the place
```

[✓ I accept the terms in the license agreement]
I do not accept the terms in the license agreement

Step-2(Agreement acceptance)



Analog Devices VisualDSP++ 3.5 for 16-Bit Processors - InstallShield W...

Custom Setup

Select the program features you want installed.

Click on an icon in the list below to change how a feature is installed.

- Blackfin Tools
  - ADSP-BF53x EZ-KIT Lite
  - Blackfin Emulators
- 21xx Tools
  - ADSP-219x EZ-KIT Lite
  - ADSP-2189 EZ-KIT Lite
  - ADSP-2181 EZ-KIT Lite
  - 21xx Emulators

Feature Description

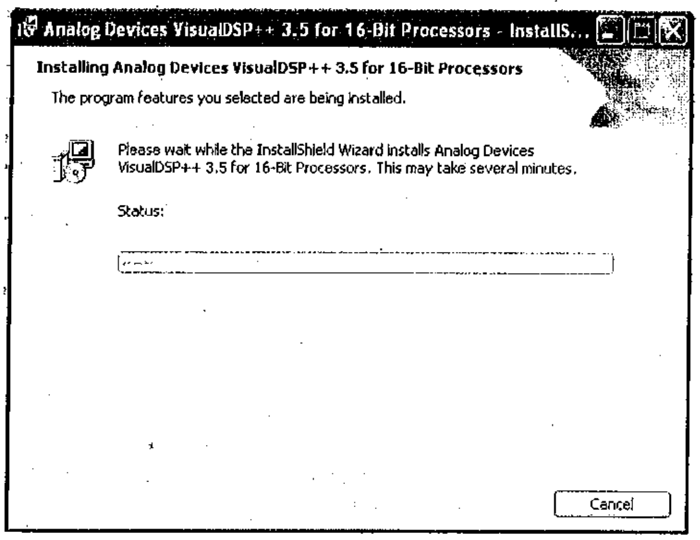
This component includes all the files neccessary to run this EZ-KIT Lite.

This feature requires 679KB on your hard drive.

Install to:

C:\Program Files\Analog Devices\VisualDSP 3.5 16-Bit\

Step-3(Choosing proper kit name)



Analog Devices VisualDSP++ 3.5 for 16-Bit Processors - InstallS...

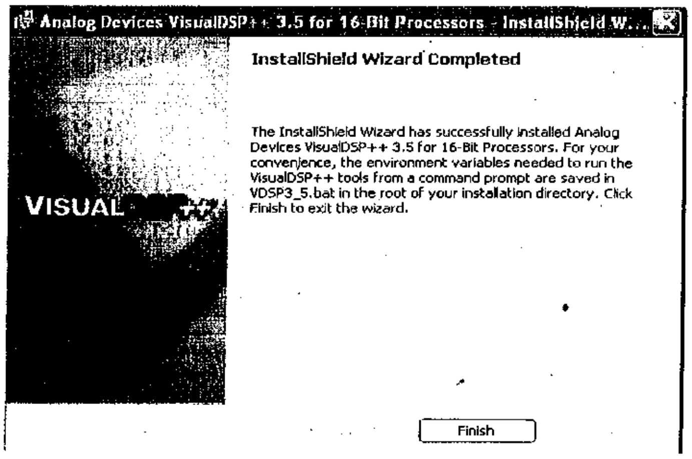
Installing Analog Devices VisualDSP++ 3.5 for 16-Bit Processors

The program features you selected are being installed.

Please wait while the InstallShield Wizard installs Analog Devices VisualDSP++ 3.5 for 16-Bit Processors. This may take several minutes.

Status:

Step-3(Further process is going on)

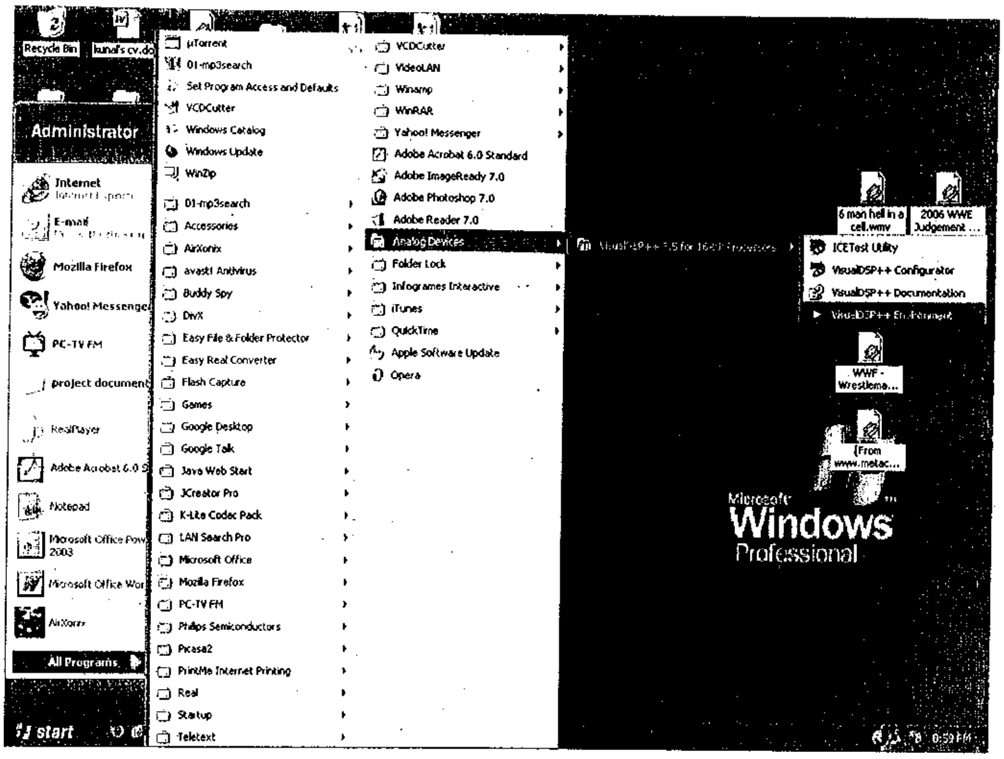


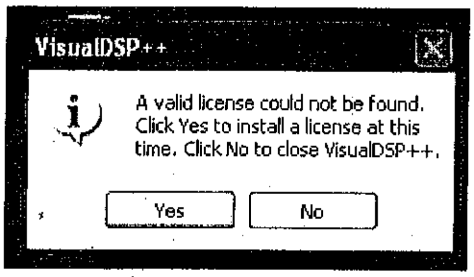
Analog Devices VisualDSP++ 3.5 for 16-Bit Processors - InstallShield W...

InstallShield Wizard Completed

The InstallShield Wizard has successfully installed Analog Devices VisualDSP++ 3.5 for 16-Bit Processors. For your convenience, the environment variables needed to run the VisualDSP++ tools from a command prompt are saved in VDSP3_5.bat in the root of your installation directory. Click Finish to exit the wizard.

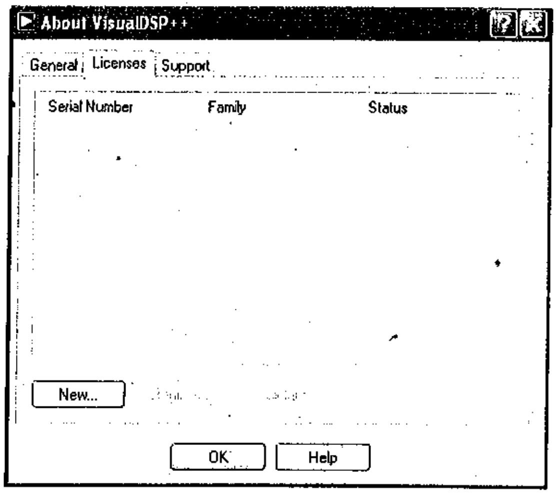
Step-4(Installation completes)

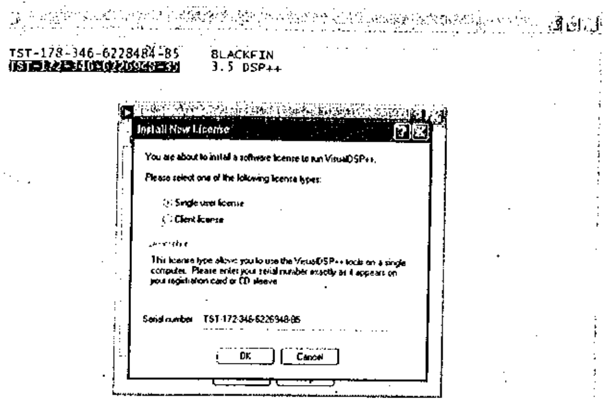




A valid license could not be found. Click Yes to install a license at this time. Click No to close VisualDSP++.

Step-5(Before running getting licence)





```
TST-178-346-6228484-85     BLACKFIN
TST-172-346-6226948-85     3.5 DSP++
```

Install New License

You are about to install a software license to run VisualDSP++.

Please select one of the following license types:

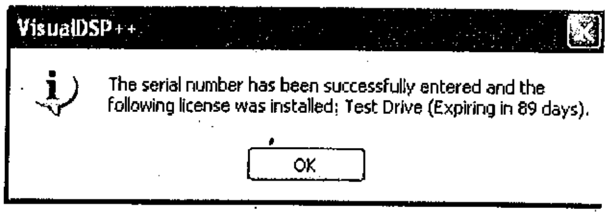
(•) Single user license
( ) Client license

Description

This license type allows you to use the VisualDSP++ tools on a single computer. Please enter your serial number exactly as it appears on your registration card or CD sleeve.

Serial number TST-172-346-6226948-85

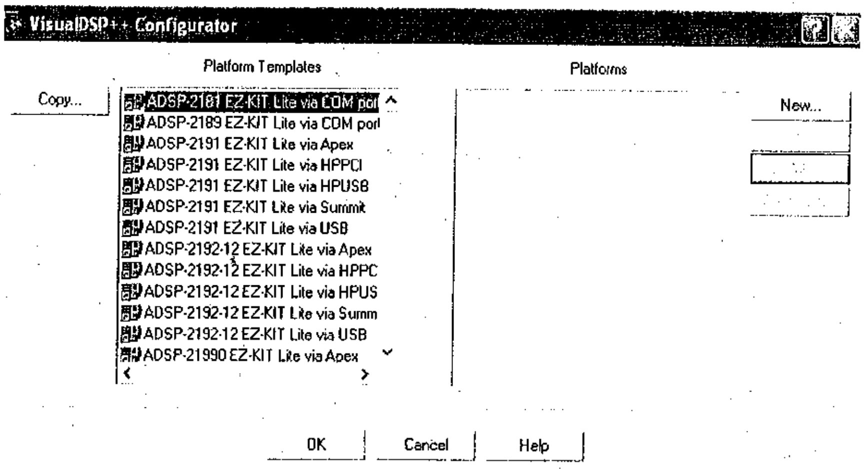
Step-6(Putting the serial key no. for trial version)



VisualDSP++

The serial number has been successfully entered and the following license was installed: Test Drive (Expiring in 89 days).

Step-7(Now we can start)



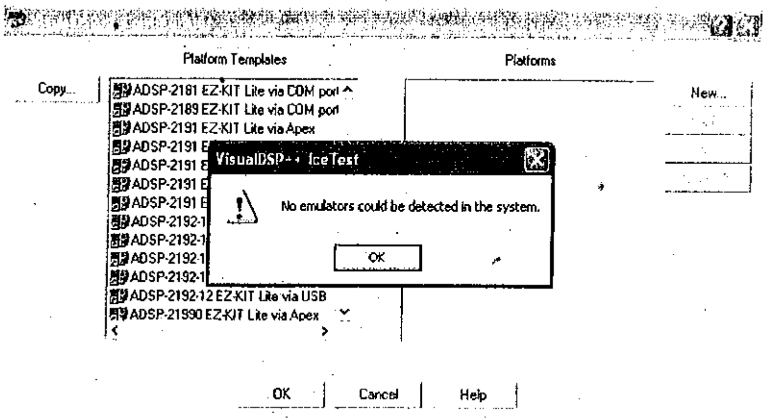
VisualDSP++ Configurator

Platform Templates

- ADSP-2181 EZ-KIT Lite via COM port
- ADSP-2189 EZ-KIT Lite via COM port
- ADSP-2191 EZ-KIT Lite via Apex
- ADSP-2191 EZ-KIT Lite via HPPCI
- ADSP-2191 EZ-KIT Lite via HPUSB
- ADSP-2191 EZ-KIT Lite via Summit
- ADSP-2191 EZ-KIT Lite via USB
- ADSP-2192-12 EZ-KIT Lite via Apex
- ADSP-2192-12 EZ-KIT Lite via HPPC
- ADSP-2192-12 EZ-KIT Lite via HPUS
- ADSP-2192-12 EZ-KIT Lite via Summ
- ADSP-2192-12 EZ-KIT Lite via USB
- ADSP-21990 EZ-KIT Lite via Apex

Platforms

Step-7(Now we will configure it by going to the configurator in program files)



Platform Templates

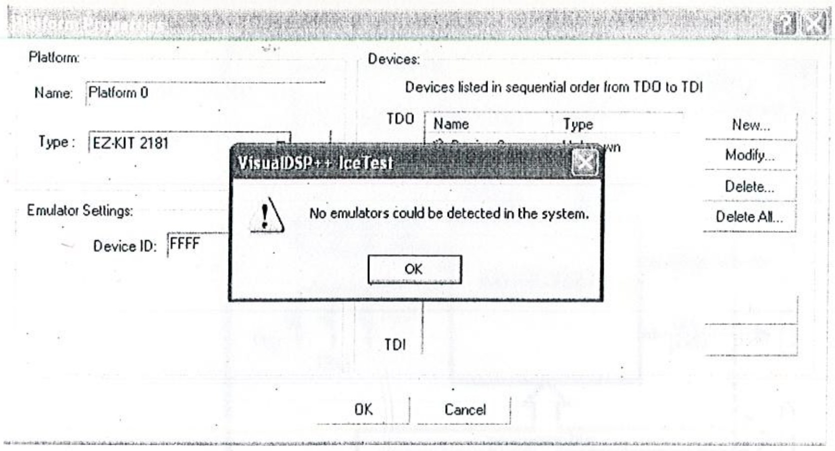
- ADSP-2181 EZ-KIT Lite via COM port
- ADSP-2189 EZ-KIT Lite via COM port
- ADSP-2191 EZ-KIT Lite via Apex
- ADSP-2192-12 EZ-KIT Lite via USB
- ADSP-21990 EZ-KIT Lite via Apex

Platforms

VisualDSP++ IceTest

No emulators could be detected in the system.

Step-8(Now we will choose the first one in configurator)


Platform:

Name: Platform 0

Type: EZ-KIT 2181

Emulator Settings:

Device ID: FFFF

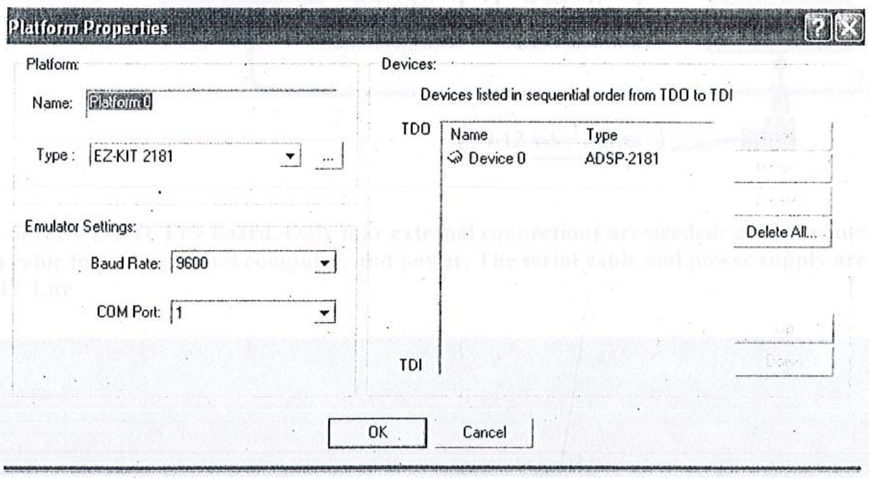
Devices:

Devices listed in sequential order from TDO to TDI

VisualDSP++ IceTest

No emulators could be detected in the system.

Step-9(Selecting EZ-KIT 2181 in type)


Platform Properties

Platform:

Name: Platform 0

Type: EZ-KIT 2181

Emulator Settings:

Baud Rate: 9600

COM Port: 1

Devices:

Devices listed in sequential order from TDO to TDI

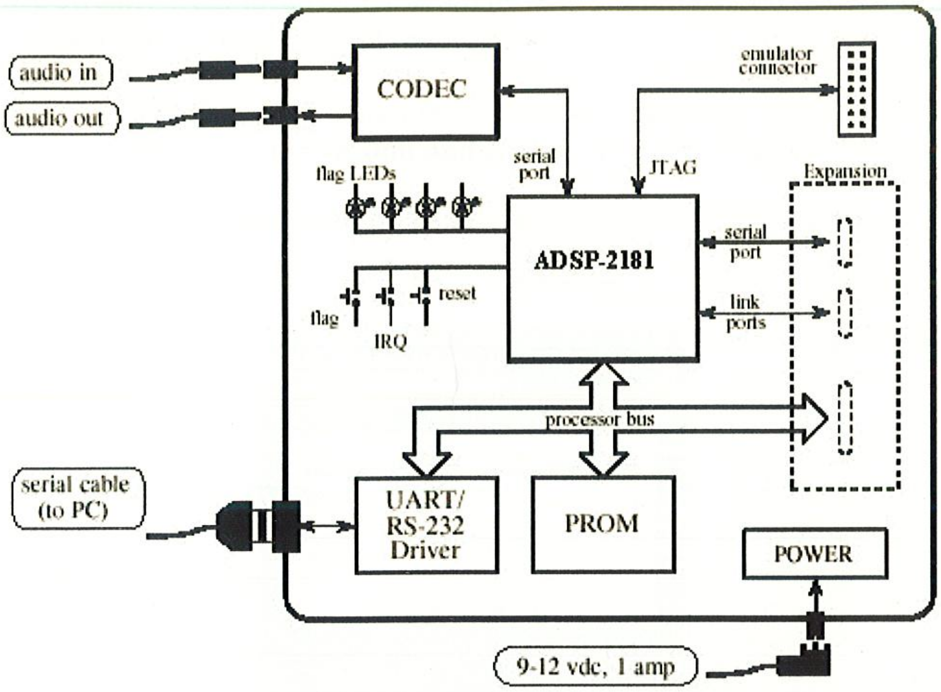
Name	Type
Device 0	ADSP-2181

Step-10(In emulator settings type 9600 in baud rate and 1 in com port)

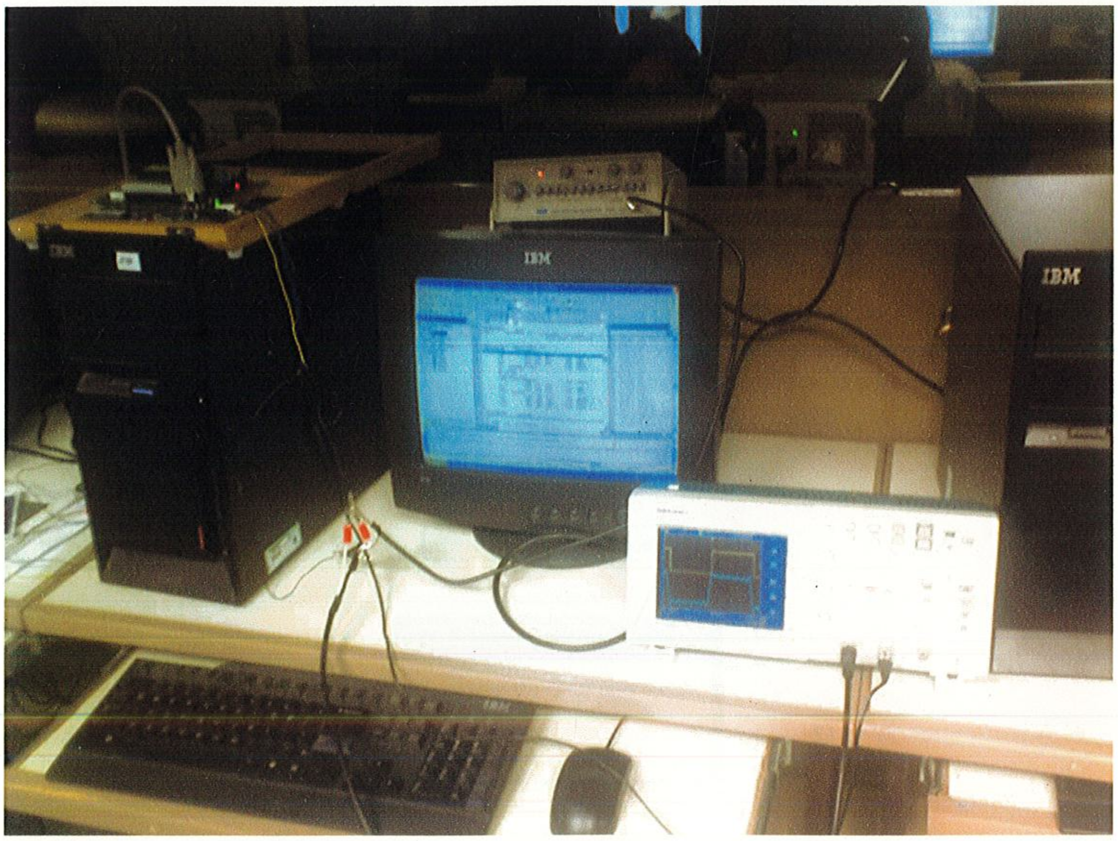
5.2.2 Interfacing to the computer:

The development cycle walks through the process, using the ADSP-2181 EZ-Kit Lite (development package ADSP-21xx- EZLITE) as the target hardware for the filter algorithm. The EZ-Kit Lite, a low-cost demonstration and development platform, consists of a 33-MHz ADSP-2181 processor, an AD1847 stereo audio codec, and a socket EPROM, which contains monitor code for downloading new algorithms to the DSP through an RS-232 connection

Now we will connect the ADSP-2181 kit with computer. Serial cable i.e.Rs-232 (9 pin cable wire) will be connected to PC. There will be power requirement of 9 to 12 volts 1amp with power port. The inputs and output will be given as shown in the diagram.



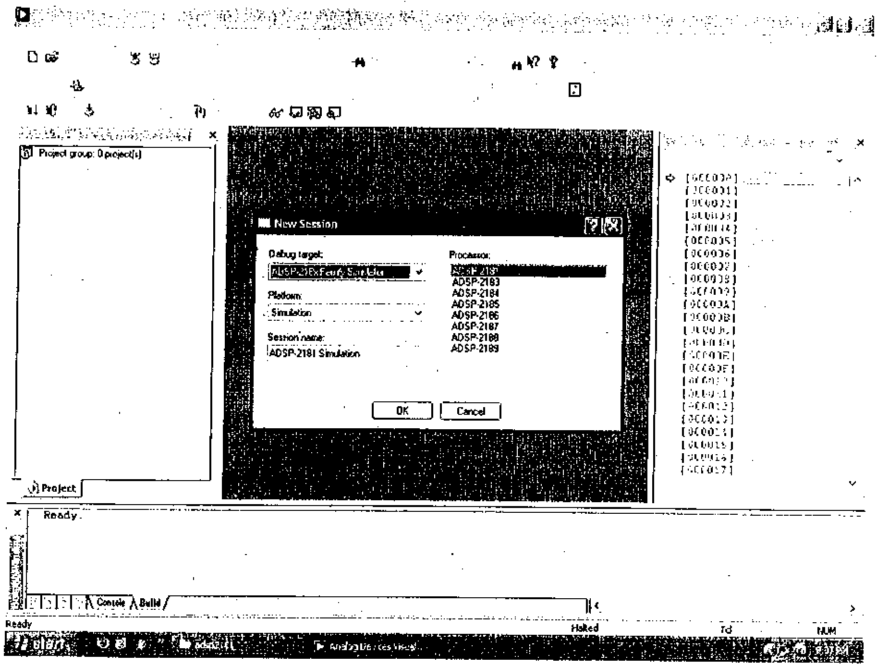
Block diagram of the EZ-KIT Lite board. Only four external connections are needed: audio in, audio out, a serial (RS-232) cable to your personal computer, and power. The serial cable and power supply are provided with the EZ-KIT Lite.



(SETUP)

If software already went into simulator mode then we have to do the following things, which we have shown as follows:-

(1) Open the Visual DSP++ environment 3.5
(2) Go to the session and click on the new session


New Session

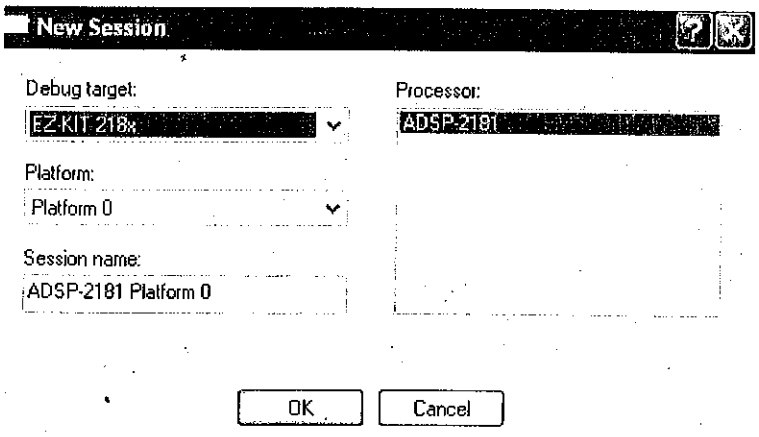
Debug target:
ADSP-218x Family Simulator

Platform:
Simulation

Session name:
ADSP-2181 Simulation

Processor:
ADSP-2181
ADSP-2183
ADSP-2184
ADSP-2185
ADSP-2186
ADSP-2187
ADSP-2188
ADSP-2189

Now in new session change the option in the debug target to ADSP-218X and then press the reset button on the kit.

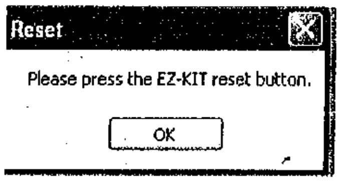

New Session

Debug target:
EZ-KIT 218x

Platform:
Platform 0

Session name:
ADSP-2181 Platform 0

Processor:
ADSP-2181


Reset

Please press the EZ-KIT reset button.

To complete the architecture description phase, one needs to know the memory and memory-mapped peripherals that the DSP has available to it. Programmers store this information in a system-description file so that the development tools software can produce appropriate code for the target system. The EZ-Kit Lite needs no memory external to the DSP, because available memory on-chip consists of the 16,384 locations of the ADSP-2181's Program Memory (PM) SRAM, and 16,352 locations of Data Memory (DM) SRAM. (32 DM locations used for system control registers are not available for working code). More information on the ADSP-2181, the EZ-Kit Lite's architecture, and related topics, can be found in texts mentioned at the end of this article.

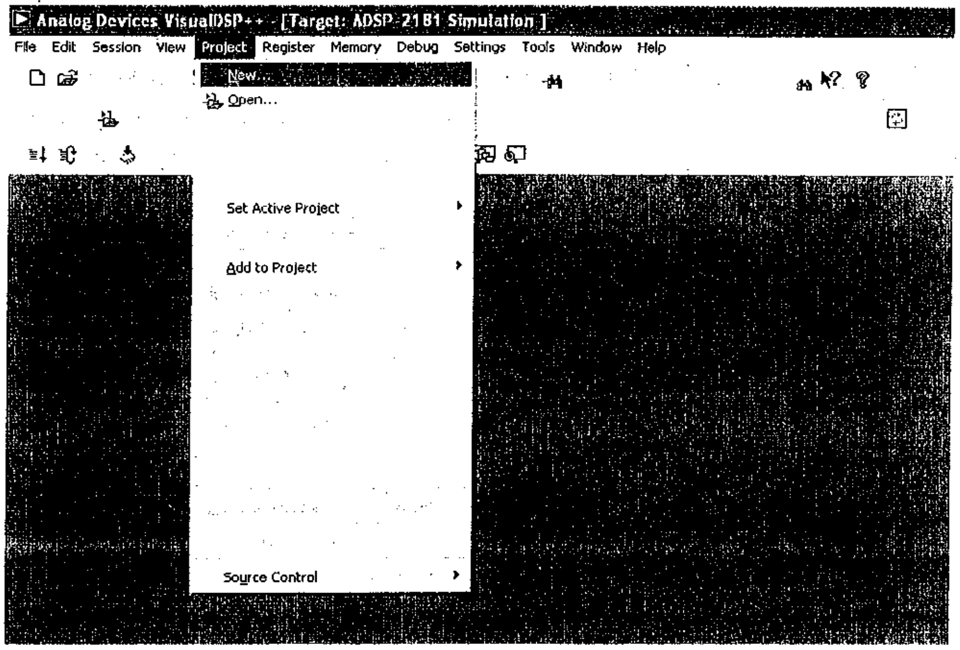
Available system resources information is recorded in a system description file for use by the ADSP-2100 Family development tools.

The listing declares 16,384 locations of PM as RAM, starting at address 0, to let both code segments and data values be placed there. Also declared are 16,352 available locations of data memory as RAM, starting at address 0. Because these processors use a Harvard architecture with two distinct memory spaces, PM address 0 is distinct from DM address 0. The ADSP-2181 EZ-Kit Lite's codec is connected to the DSP using a serial port, which is not declared in the system description file. To make the system description file available to other software tools, the System Builder utility, BLD21, converts the .SYS file into an architecture, or .ACH, file. The output of the System Builder is a file named EZKIT_LT.ACH.

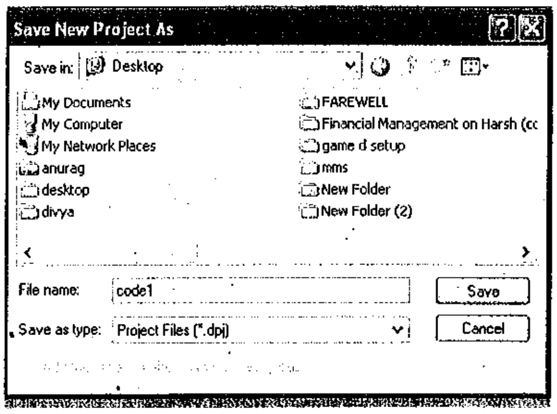
After writing the code, the next step is to generate an executable file, i.e., turn the code into instructions that the DSP can execute. First one assembles the DSP code. This converts the program file into a format that the other development tools can process. Assembling also checks the code for syntax errors. Next, one links the code to generate the DSP executable, using the available memory that is declared in the architecture file. The Linker fits all of the code and data from the source code into the memory space; the output is a DSP executable file, which can be downloaded to the EZ-Kit Lite board.

5.2.3 How to run the project file on the system:

The following diagrams show how to run project on the system.


Analog Devices VisualDSP++ - [Target: ADSP-2181 Simulation]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
New...
Open...
Set Active Project
Add to Project
Source Control

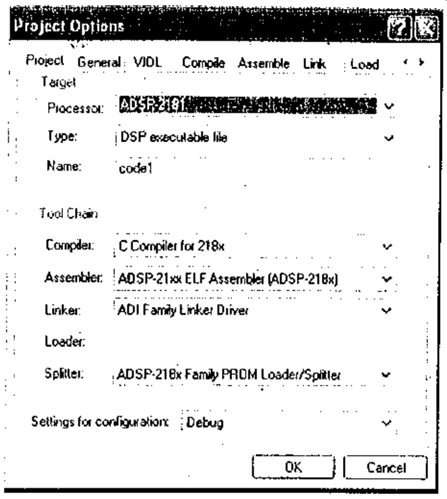
Step-1(Open new in project option)



File name: code1

Save as type: Project Files (*.dpj)

Step-2(Now type a name for the project)


Project Options

Project | General | VIDL | Compile | Assemble | Link | Load

Target

Processor: ADSP-2181

Type: DSP executable file

Name: code1

Tool Chain

Compiler: C Compiler for 218x

Assembler: ADSP-21xx ELF Assembler (ADSP-218x)

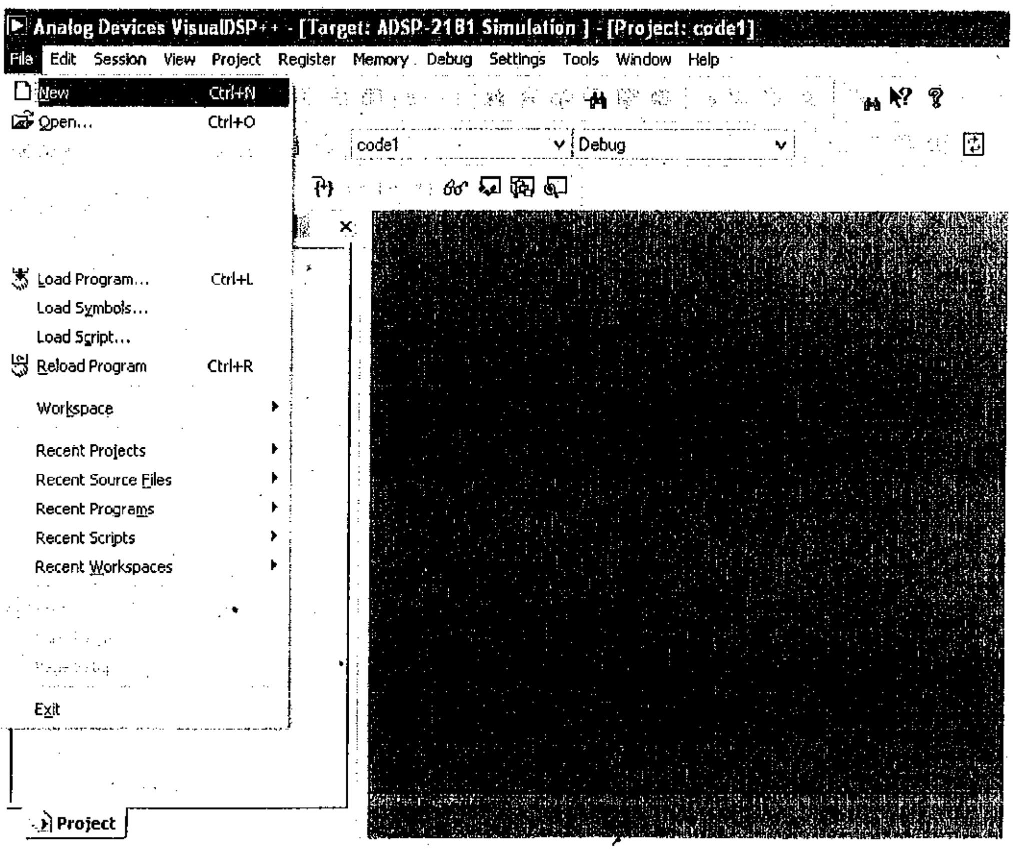
Linker: ADI Family Linker Driver

Loader:

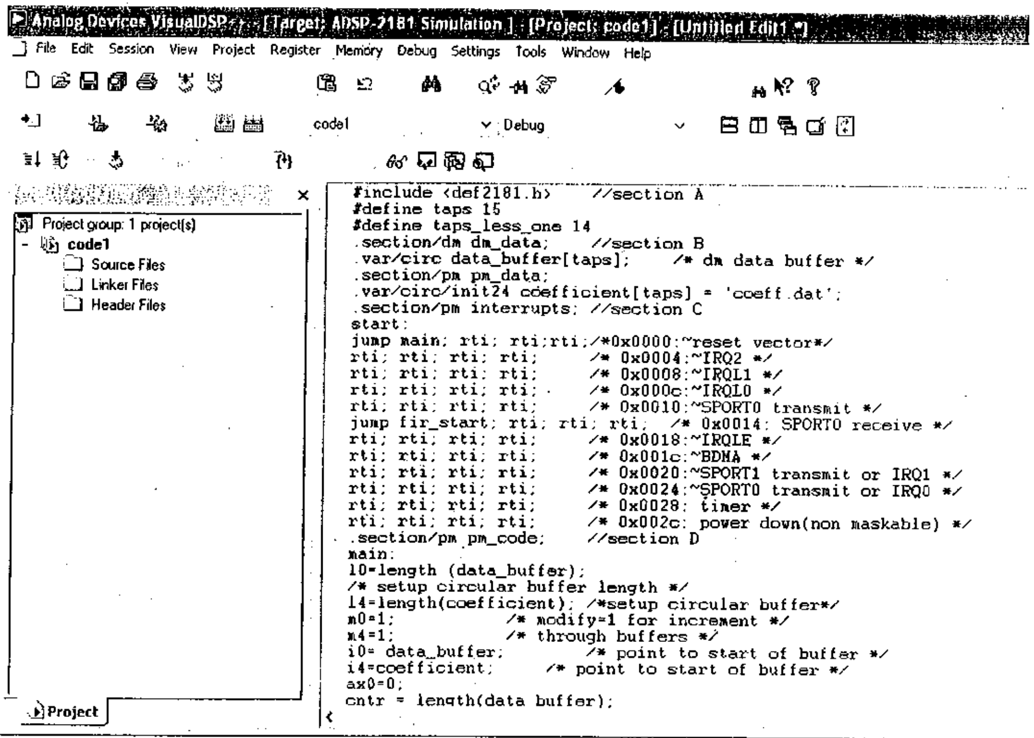
Splitter: ADSP-218x Family PROM Loader/Splitter

Settings for configuration: Debug

Step-3(Now click ok in project option)

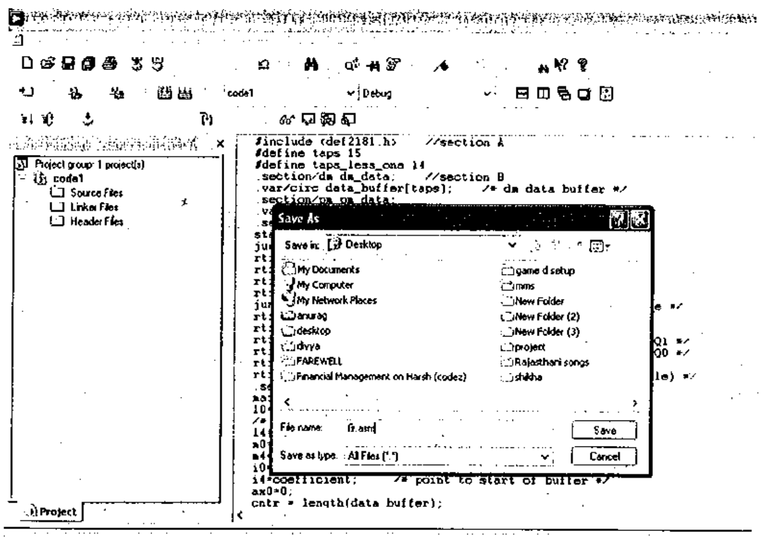

Analog Devices VisualDSP++ - [Target: ADSP-2181 Simulation] - [Project: code1]

Step-4(Click on file option and open a new file)


Analog Devices VisualDSP++ - [Target: ADSP-2181 Simulation] - [Project: code1] - [Untitled Edit1 *]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
code1
Debug
Project group: 1 project(s)
code1
Source Files
Linker Files
Header Files
Project

```
#include <def2181.h>     //section A
#define taps 15
#define taps_less_one 14
.section/dm dm_data;     //section B
.var/circ data_buffer[taps];     /* dm data buffer */
.section/pm pm_data;
.var/circ/init24 coefficient[taps] = 'coeff.dat';
.section/pm interrupts; //section C
start:
jump main; rti; rti;rti;/*0x0000:~reset vector*/
rti; rti; rti; rti;      /* 0x0004:~IRQ2 */
rti; rti; rti; rti;      /* 0x0008:~IRQL1 */
rti; rti; rti; rti;      /* 0x000c:~IRQL0 */
rti; rti; rti; rti;      /* 0x0010:~SPORT0 transmit */
jump fir_start; rti; rti; rti;  /* 0x0014: SPORT0 receive */
rti; rti; rti; rti;      /* 0x0018:~IRQLE */
rti; rti; rti; rti;      /* 0x001c:~BDMA */
rti; rti; rti; rti;      /* 0x0020:~SPORT1 transmit or IRQ1 */
rti; rti; rti; rti;      /* 0x0024:~SPORT0 transmit or IRQ0 */
rti; rti; rti; rti;      /* 0x0028: timer */
rti; rti; rti; rti;      /* 0x002c: power down(non maskable) */
.section/pm pm_code;     //section D
main:
l0=length (data_buffer);
/* setup circular buffer length */
l4=length(coefficient); /*setup circular buffer*/
m0=1;              /* modify=1 for increment */
m4=1;              /* through buffers */
i0= data_buffer;          /* point to start of buffer */
i4=coefficient;      /* point to start of buffer */
ax0=0;
cntr = length(data buffer);
```

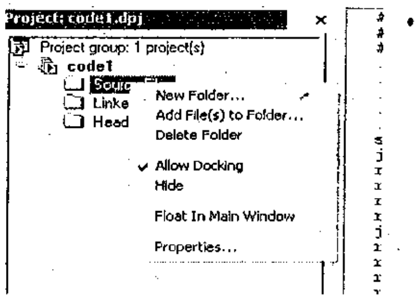
Step-5(Now paste the written code in new file)

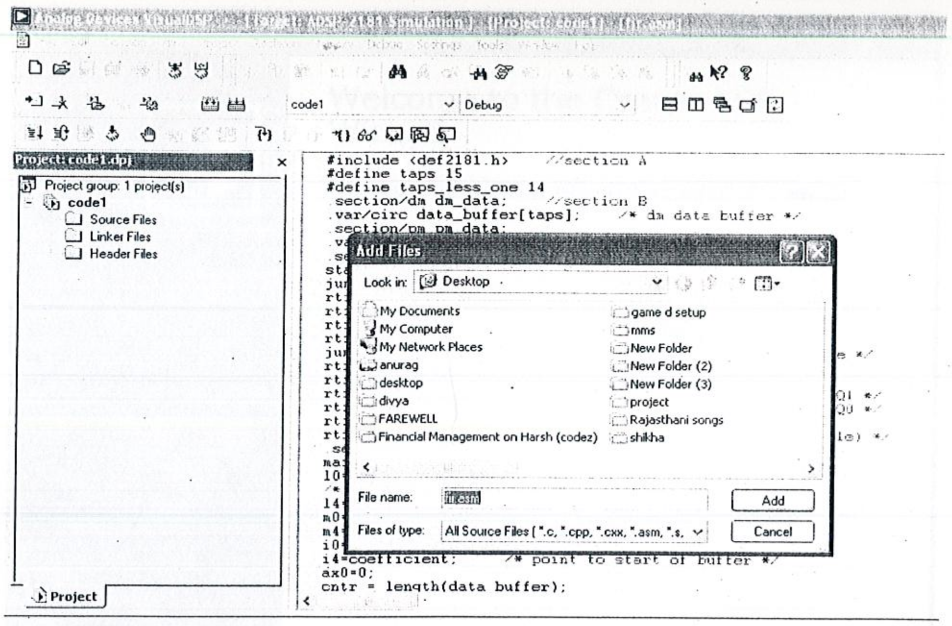


File name: fir.asm

Save as type: All Files (*.*)

Step-6(Save thė code as .asm, :c, c++ as per requirement)



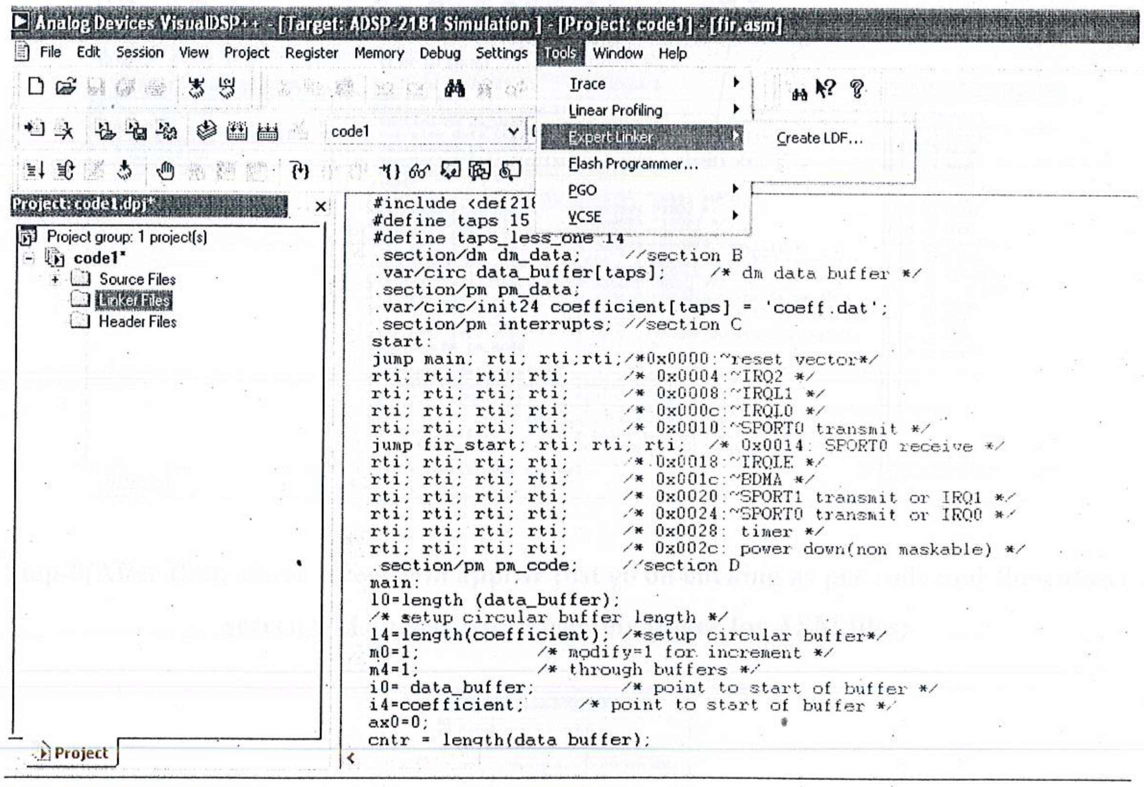


File name: fir.asm

Files of type: All Source Files (*.c, *.cpp, *.cxx, *.asm, *.s,

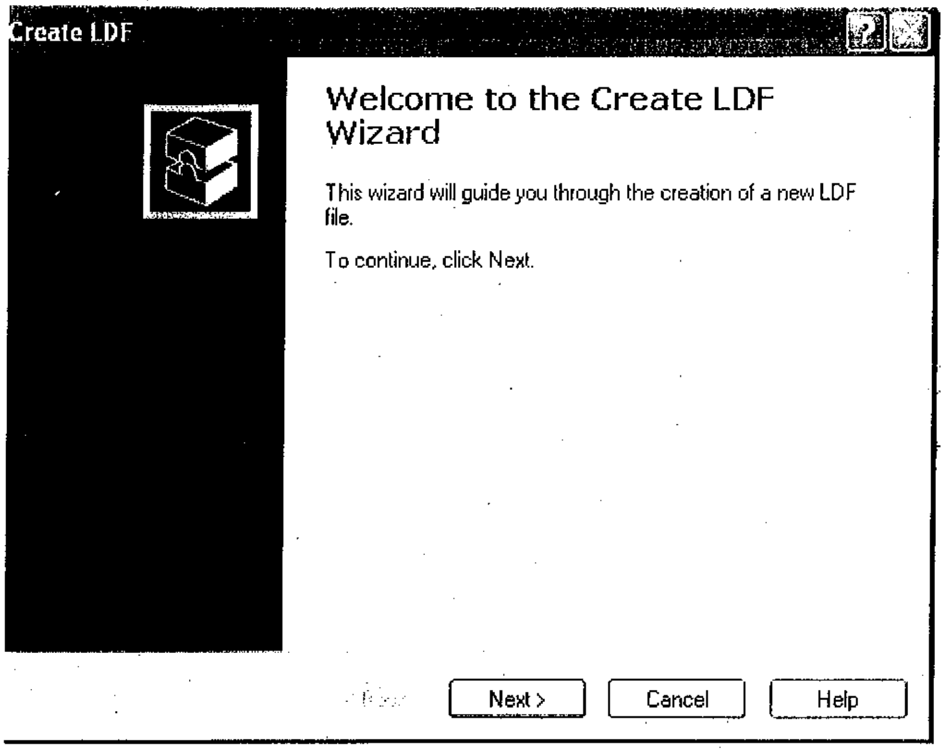
```
#include <def2181.h>     //section A
#define taps 15
#define taps_less_one 14
 section/dm dm_data;     //section B
.var/circ data_buffer[taps];     /* dm data buffer */
 section/pm pm_data;
i4=coefficient;        /* point to start of buffer */
ax0=0;
cntr = length(data buffer);
```

Step-7(Now add above file to the source folder)



```
#include <def218
#define taps 15
#define taps_less_one 14
.section/dm dm_data;     //section B
.var/circ data_buffer[taps];     /* dm data buffer */
.section/pm pm_data;
.var/circ/init24 coefficient[taps] = 'coeff.dat';
.section/pm interrupts; //section C
start:
jump main; rti; rti;rti;/*0x0000:~reset vector*/
rti; rti; rti; rti;      /* 0x0004:~IRQ2 */
rti; rti; rti; rti;      /* 0x0008:~IRQL1 */
rti; rti; rti; rti;      /* 0x000c:~IRQL0 */
rti; rti; rti; rti;      /* 0x0010:~SPORT0 transmit */
jump fir_start; rti; rti; rti;  /* 0x0014: SPORT0 receive */
rti; rti; rti; rti;      /* 0x0018:~IRQLE */
rti; rti; rti; rti;      /* 0x001c:~BDMA */
rti; rti; rti; rti;      /* 0x0020:~SPORT1 transmit or IRQ1 */
rti; rti; rti; rti;      /* 0x0024:~SPORT0 transmit or IRQ0 */
rti; rti; rti; rti;      /* 0x0028: timer */
rti; rti; rti; rti;      /* 0x002c: power down(non maskable) */
.section/pm pm_code;     //section D
main:
l0=length (data_buffer);
/* setup circular buffer length */
l4=length(coefficient); /*setup circular buffer*/
m0=1;            /* modify=1 for increment */
m4=1;            /* through buffers */
i0= data_buffer;           /* point to start of buffer */
i4=coefficient;        /* point to start of buffer */
ax0=0;
cntr = length(data buffer);
```

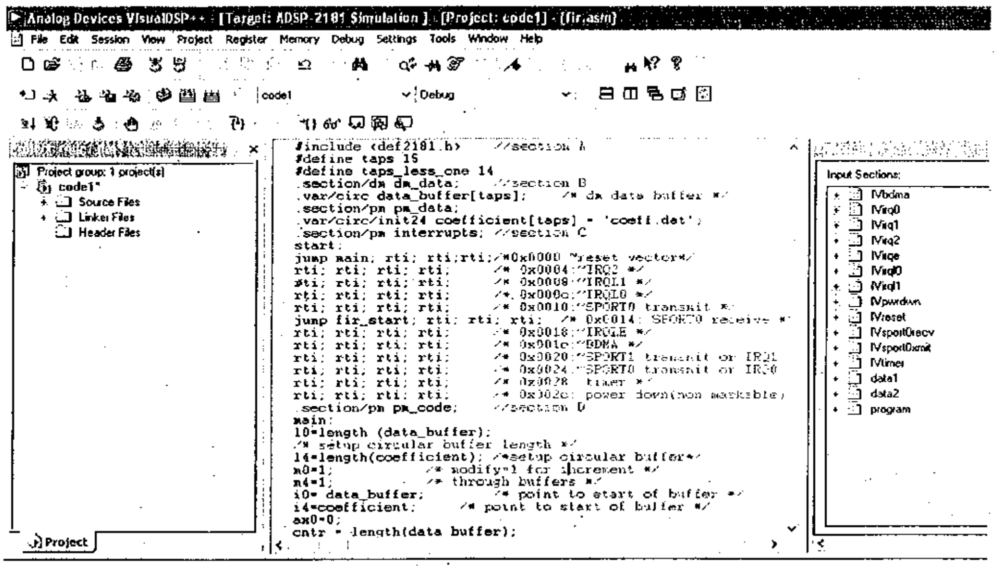
Step-8(Now for creating linker file go to tools and click on expert linker and create LDF file)


Create LDF

Welcome to the Create LDF Wizard

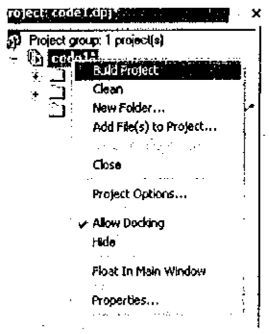
This wizard will guide you through the creation of a new LDF file.

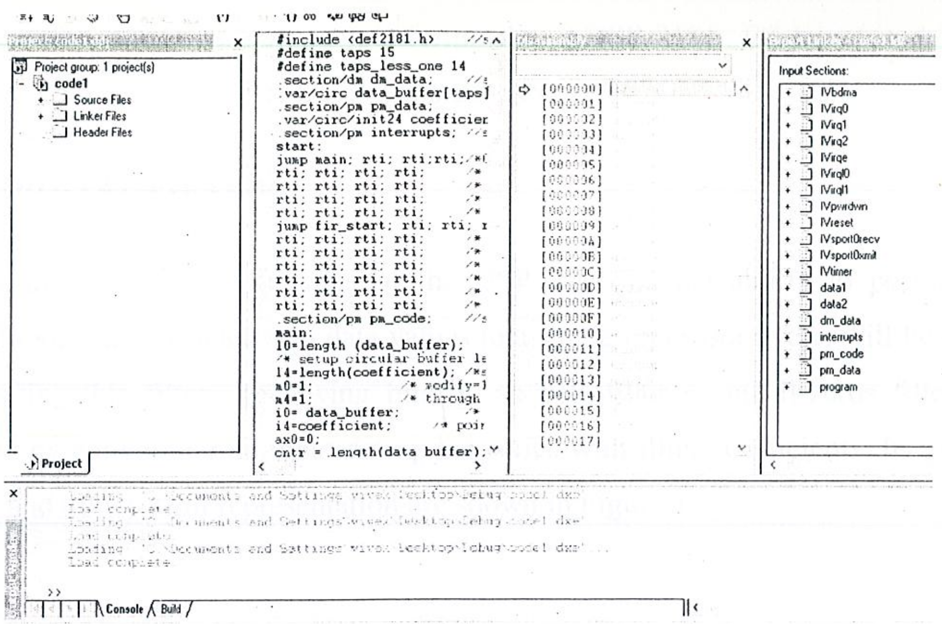
To continue, click Next.


Analog Devices VisualDSP++ - [Target: ADSP-2181 Simulation] - [Project: code1] - [fir.asm]
File Edit Session View Project Register Memory Debug Settings Tools Window Help
code1
Debug
Project group: 1 project(s)
code1*
Source Files
Linker Files
Header Files
Project
Input Sections:
IVbdma
IVirq0
IVirq1
IVirq2
IVirqe
IVirql0
IVirql1
IVpwrdwn
IVreset
IVsport0recv
IVsport0xmit
IVtimer
data1
data2
program

```
#include <def2181.h>      //section A
#define taps 15
#define taps_less_one 14
.section/dm dm_data;      //section B
.var/circ data_buffer[taps];      /* dm data buffer */
.section/pm pm_data;
.var/circ/init24 coefficient[taps] = 'coeff.dat';
.section/pm interrupts; //section C
start:
jump main; rti; rti;rti;/*0x0000 ~reset vector*/
rti; rti; rti; rti;       /* 0x0004:~IRQ2 */
rti; rti; rti; rti;       /* 0x0008:~IRQL1 */
rti; rti; rti; rti;       /* 0x000c:~IRQL0 */
rti; rti; rti; rti;       /* 0x0010:~SPORT0 transmit */
jump fir_start; rti; rti; rti;  /* 0x0014: SPORT0 receive */
rti; rti; rti; rti;       /* 0x0018:~IRQE */
rti; rti; rti; rti;       /* 0x001c:~BDMA */
rti; rti; rti; rti;       /* 0x0020:~SPORT1 transmit or IRQ1
rti; rti; rti; rti;       /* 0x0024:~SPORT0 transmit or IRQ0
rti; rti; rti; rti;       /* 0x0028  timer */
rti; rti; rti; rti;       /* 0x002c: power down(non maskable)
.section/pm pm_code;      //section D
main:
l0=length (data_buffer);
/* setup circular buffer length */
l4=length(coefficient); /*setup circular buffer*/
m0=1;             /* modify=1 for increment */
m4=1;             /* through buffers */
i0= data_buffer;            /* point to start of buffer */
i4=coefficient;      /* point to start of buffer */
ax0=0;
cntr = length(data buffer);
```

Step-9(After that, above screen will appear just go on clicking as per code and then above screen will appear with assembly linker for ASM files)

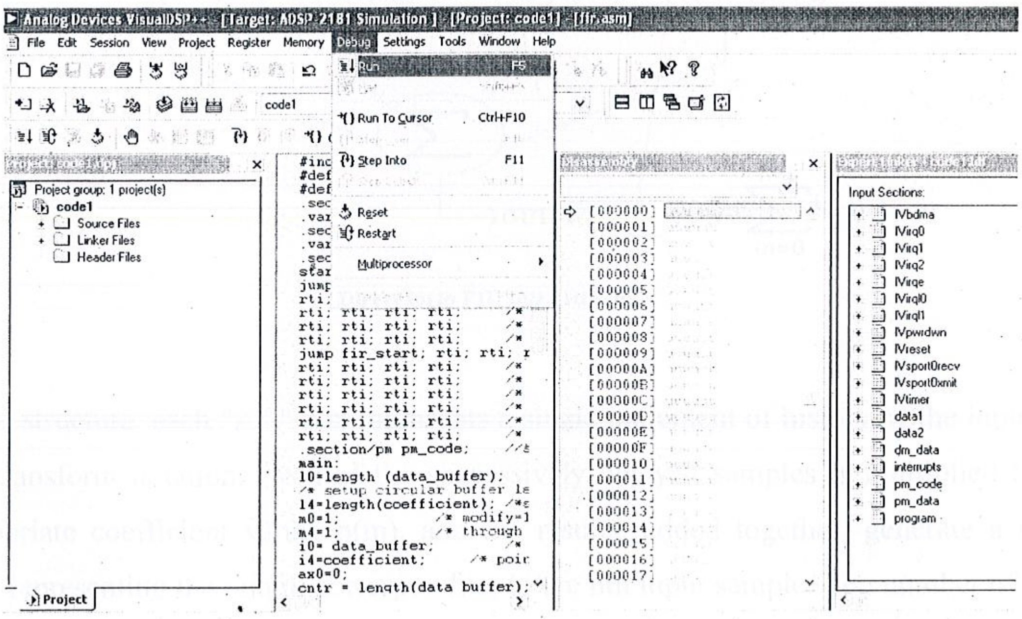




```
#include <def2181.h>          //s
#define taps 15
#define taps_less_one 14
.section/dm dm_data;          //s
.var/circ data_buffer[taps]
.section/pm pm_data;
.var/circ/init24 coefficier
.section/pm interrupts;       //s
start:
jump main; rti; rti;rti;/*(
rti; rti; rti; rti;           /*
rti; rti; rti; rti;           /*
rti; rti; rti; rti;           /*
rti; rti; rti; rti;           /*
jump fir_start; rti; rti; r
rti; rti; rti; rti;           /*
rti; rti; rti; rti;           /*
rti; rti; rti; rti;           /*
rti; rti; rti; rti;           /*
rti; rti; rti; rti;           /*
rti; rti; rti; rti;           /*
.section/pm pm_code;          //s
main:
l0=length (data_buffer);
/* setup circular buffer le
l4=length(coefficient); /*s
m0=1;                /* modify=1
m4=1;                /* through
i0= data_buffer;              /*
i4=coefficient;         /* poir
ax0=0;
cntr = length(data buffer);
```

```
Loading 'C:\Documents and Settings\vivek\Desktop\Debug\code1.dxe'
Load complete.
Loading 'C:\Documents and Settings\vivek\Desktop\Debug\code1.dxe'
Load complete.
Loading 'C:\Documents and Settings\vivek\Desktop\Debug\code1.dxe'...
Load complete.

>>
```

Step-10(Now right click to build project and then it will show that load is complete or not or it may have errors or not)



Analog Devices VisualDSP++ - [Target: ADSP-2181 Simulation] - [Project: code1] - [fir.asm]

```
#inc
#def
#def
.sec
.var
.sec
.var
.sec
star
jump
rti;
rti; rti; rti; rti;           /*
rti; rti; rti; rti;           /*
rti; rti; rti; rti;           /*
jump fir_start; rti; rti; r
rti; rti; rti; rti;           /*
rti; rti; rti; rti;           /*
rti; rti; rti; rti;           /*
rti; rti; rti; rti;           /*
rti; rti; rti; rti;           /*
rti; rti; rti; rti;           /*
.section/pm pm_code;          //s
main:
l0=length (data_buffer);
/* setup circular buffer le
l4=length(coefficient); /*s
m0=1;                /* modify=1
m4=1;                /* through
i0= data_buffer;              /*
i4=coefficient;         /* poir
ax0=0;
cntr = length(data buffer);
```

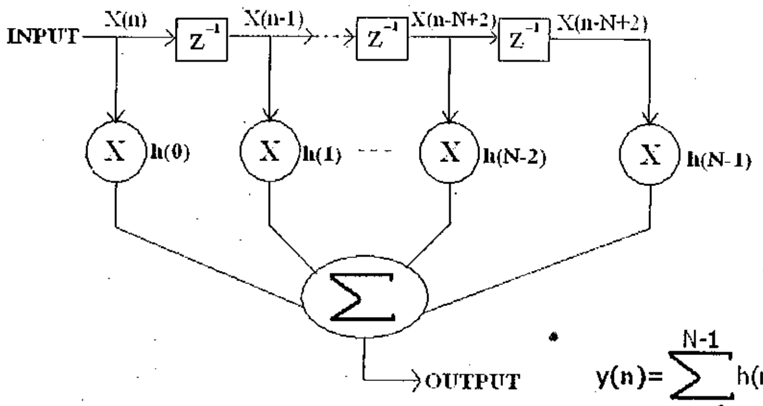
Step-11(Finally we can run the project from Debug option)

5.3 Implementing Algorithms on a Hardware Platform:

So far, we have described the physical architecture of the DSP processor, explained how DSP can provide some advantages over traditionally analog circuitry, and examined digital filtering, showing how the programmable nature of DSP lends itself to such algorithms. Now we look at the process of implementing a finite-impulse- response (FIR) filter algorithm on a hardware platform, the ADSP-2181 EZ-Kit Lite&tm;. The implementation is expanded to handle data I/O issues.

USING DIGITAL FILTERS:

Many of the architectural features of the DSP, such as the ability to perform zero-overhead loops, and to fetch two data values in a single processor cycle, will be useful in implementing this filter. Reviewing briefly, an FIR filter is an all-zeros filter that is calculated by convolving an input data-point series with filter coefficients. Its governing equation and direct-form representation are shown in Figure 1.



$$y(n)=\sum_{m=0}^{N-1} h(m)x(n-m)$$

Direct form FIR filter structure

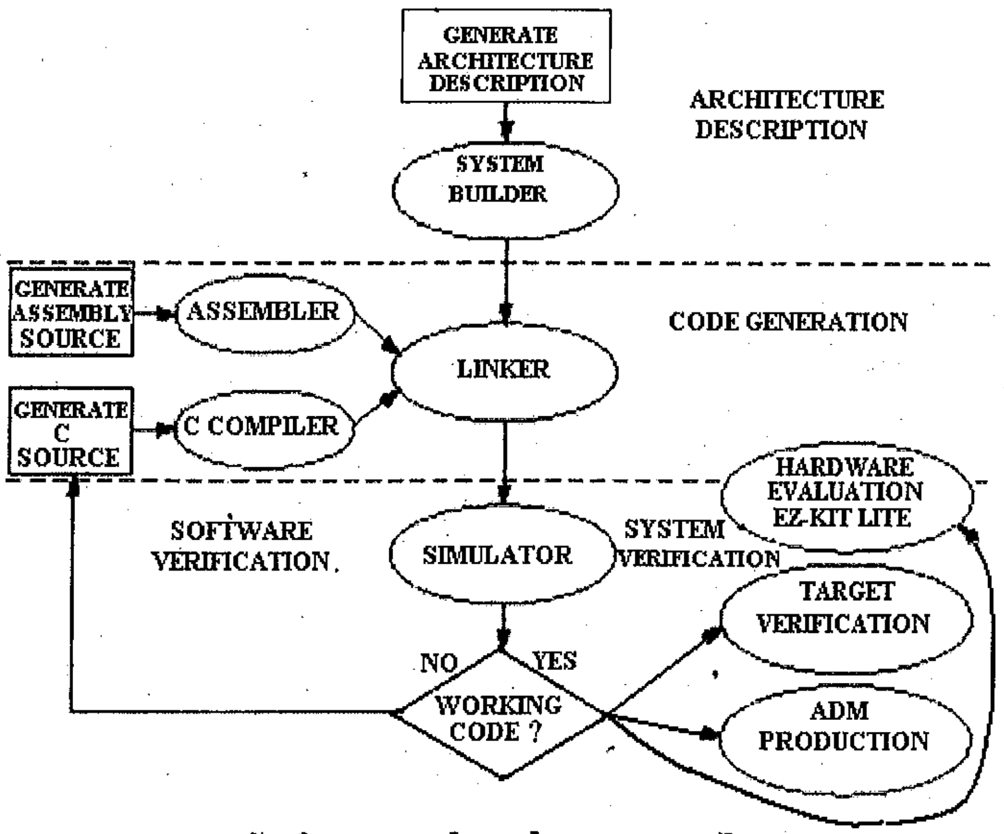
In this structure, each "z^{-1} " box represents a single increment of history of the input data in z-transform notation. Each of the successively delayed samples is multiplied by the appropriate coefficient value, h(m), and the results, added together, generate a single value representing the output corresponding to the nth input sample. The number of delay elements, or filter taps, and their coefficient values, determine the filter's performance. The filter structure suggests the physical elements needed to implement this algorithm by computation using a DSP. For the computation itself, each output sample requires a number of multiply-accumulate operations equal to the length of the filter. The delay line for input data and the coefficient value list require reserved areas of memory in the DSP for storing data values and coefficients. The DSP's enhanced Harvard architecture lets programmers store data in Program Memory as well as in Data Memory, and thus perform two simultaneous memory accesses in every cycle from the DSP's internal

SRAM. With Data Memory holding the incoming samples, and Program Memory storing the coefficient values, both a data value and a coefficient value can be fetched in a single cycle for computation.

This DSP architecture favors programs that use circular buffering. The implication is that address pointers need to be initialized only at the beginning of the program, and the circular buffering mechanism ensures that the pointer does not leave the bounds of its assigned memory buffer—a capability used extensively in the FIR filter code for both input delay line and coefficients. Once the elements of the program have been determined, the next step is to develop the DSP source code to implement the algorithm.

DEVELOPING DSP SOFTWARE:

Software development flow for the ADSP-2100 Family consists of the following steps: architecture description, source-code generation, software validation (debugging), and hardware implementation. Figure 2 shows a typical development cycle.



Software development flow

Source-code generation:

Moving from theory into practice, this step—where an algorithmic idea is turned into code that runs on the DSP—is often the most time-consuming step in the process. There are several ways to generate source code. Some programmers prefer to code their algorithms in a high-level language such as C; others prefer to use the processor's native assembly language. Implementations in C may be faster for the programmer to develop, but compiled DSP code lacks efficiency by not taking full advantage of a processor's architecture.

Assembly code, by taking full advantage of a processor's design, yields highly efficient implementations. But the programmer needs to become familiar with the processor's native assembly language. Most effective is combining C for high-level program-control functions and assembly code for the time-critical, math-intensive portions of the system. In any case, the programmer must be aware of the processor's system constraints and peripheral specifics. The FIR filter system example in this article uses the native assembly language of the ADSP-2100 Family.

Software validation ("debugging"):

This phase tests the results of code generation—using a software tool known as a simulator— to check the logical flow of the program and verify that an algorithm is performing as intended. The simulator is a model of the DSP processor that a) provides visibility into all memory locations and processor registers, b) allows the user to run the DSP code either continuously or one instruction at a time, and c) can simulate external devices feeding data to the processor.

Hardware implementation:

Here the code is run on a real DSP, typically in several phases: a) tryout on an evaluation platform such as EZ-Kit Lite; b) in-circuit emulation, and c) production ROM generation. Tryout provides a quick go/no-go determination of the program's operation; this technique is the implementation method used in this article. In-circuit emulation monitors software debug in the system, where a tool such as an EZ-ICE™ controls processor operation on the target platform. After all debug is complete, a boot ROM of the final code can be generated; it serves as the final production implementation.

GENERATING FILTER CODE:

The code is augmented to incorporate some EZ-Kit Lite-specific features, specifically codec initialization and data I/O. The core filter-algorithm elements (multiply-accumulates, data addressing using circular buffers for both data and coefficients, and reliance on the efficiency of the zero-overhead loop) do not change.

The incoming data will be sampled using the on-board AD1847 codec, which has programmable sampling rate, input gain, output attenuation, input selection, and input mixing. Its programmable nature makes the system flexible, but it also adds a task of programming to initialize it for the DSP system.

ACCESSING DATA:

For this example, a series of control words to the codec—to be defined at the beginning of the program in the first section of the listing—will initialize it for an 8-kHz sampling rate, with moderate gain values on each of the input channels. Since the AD1847 is programmable, users would typically reuse interface and initialization code segments, changing only the specific register values for different applications. This example will add the specific filter segment to an existing code segment found in the EZ-Kit Lite software.

This interface code declares two areas in memory to be used for data I/O: "tx_buf", for data to be transmitted out of the codec, and "rx_buf", where incoming data is received. Each of these memory areas, or buffers, contains three elements, a control or status word, left-channel data, and right-channel data. For each sample period, the DSP will receive from the codec a status word, left channel data, and right channel data. On every sample period, the DSP must supply to the codec a transmit control word, left channel data, and right channel data. In this application, the control information sent to the codec will not be altered, so the first word in the transmit data buffer will be left as is. We will assume that the source is a monophonic microphone, using the right channel (no concern about left-channel input data).

Using the I/O shell program found in the EZ-Kit Lite software, we need only be involved with the section of code labeled "input_samples". This section of code is accessed when new data is received from the codec ready to be processed. If only the right channel data is required, we need to read the data located in data memory at location rx_buf + 2, and place it in a data register to be fed into the filter program.

The data arriving from the codec needs to be fed into the filter algorithm via the input delay line, using the circular buffering capability of the ADSP-2181. The length of the input delay line is determined by the number of coefficients used for the filter. Because the data buffer is circular, the oldest data value in the buffer will be wherever the pointer is pointing after the last filter access. Likewise the coefficients, always accessed in the same order every time through the filter, are placed in a circular buffer in Program Memory.

Algorithm

To operate on the received data, the code section published in the last installment can be used with few modifications. To implement this filter, we need to use the multiply/accumulate (MAC) computational unit and the data address-generators.

The ADSP-2181's MAC stores the result in a 40-bit register (32 bits for the product of 2 16-bit words, and 8 bits to allow the sum to expand without overflowing). This allows intermediate filter values to grow and shrink as necessary without corrupting data. The code segment being used is generic (i.e., can be used for any length filters); so the MAC's extra output bits allow arbitrary filters with unknown data to be run with little fear of losing data.

To implement the FIR filter, the multiply/accumulate operation is repeated for all taps of the filter on each data point. To do this (and be ready for the next data point), the MAC instruction is written in the form of a loop. The ADSP-21xx's zero-overhead loop capability allows the MAC instruction to be repeated for a specified number of counts without programming intervention. A counter is set to the number of taps minus one, and the loop mechanism automatically decrements the counter for each loop operation. Setting the loop counter to "taps–1" ensures that the data pointers end up in the correct location after execution is finished and allows the final MAC operation to include rounding. As the AD1847 is a 16-bit codec, the MAC with rounding provides a statistically unbiased result rounded to the nearest 16-bit value. This final result is written to the codec.

For optimal code execution, every instruction cycle should perform a meaningful mathematical calculation. The ADSP-21xxs accomplish this with multi-function instructions: the processor can perform several functions in the same instruction cycle. For the FIR filter code, each multiply-accumulate (MAC) operation can be performed in parallel with two data accesses, one from Data Memory, one from Program Memory. This capability means that on every loop iteration a MAC operation is being performed. At the same time, the next data value and coefficient are being fetched, and the counter is automatically decremented. All without wasting time maintaining loops.

As the filter code is executed for each input data sample, the output of the MAC loop will be written to the output data buffer, tx_buf. Although this program only deals with single-channel input data, the result will be written out to both channels by writing to memory buffer addresses tx_buf+1 and tx_buf+2.

```
/*--------------------FIR Filter--------------------*/
input_samples:
   ena sec_reg;            /* use shadow register bank */

/* set up for filter 1 */
i2 = dm(filter1_ptr);     /* set data pointer for filter 1
*/
ax0 = dm(rx_buf + 1);     /* read left channel data */
dm(i2,m1) = ax0;    /* write new data into delay line,
                     pointer now pointing to oldest data
                      . */
call filter;         /* perform the first filter for left
                       Channel data */

dm(tx_buf+1) = mr1;  /* write left-channel output data */
dm(filter1_ptr) = i2;     /* save updated filter1 data
                            pointer */
```

```
/* set up for filter 2 */
i2 = dm(filter2_ptr);      /* set data pointer for filter 2
*/
ax0 = dm(rx_buf + 2);      /* read right channel data */
dm(i2,m1) = ax0;          /* write new data into delay line,
                             Pointer now pointing to oldest
data */

Call filter;           /* perform the filter again for the
                          Right channel data */

dm(tx_buf+2) = mr1;/* write right channel output data */
dm(filter2_ptr) = i2;     /* save updated filter2 data
pointer */

rti;                    /* return from interrupt */
```

The final source code listing is shown as above. The filter algorithm itself is listed under "Interrupt service routines". The rest of the code is used for codec and DSP initialization and interrupt service routine definition. Those topics will be explored in future installments of this series.

The following listing shows how the FIR Filter interrupt routine uses these new memory elements. The original Filter subroutine from the 3rd installment has been modified to provide two separate channels of filtering. Instead of launching directly into the filter calculation, the routine must first load the appropriate data pointer. The filter routine is then called, and the resulting output is placed in the correct location for transmission.

Because the core filter algorithm no longer handles data I/O, this subroutine can be expanded to more channels of filtering by merely adding more pointer variables and declaring more buffer space (as long as sufficient memory exists!) Similarly, different coefficients can be used for the two filters by setting up variables that contain coefficient-buffer pointer information. In either case, the filter algorithm does not need to be altered.

By using this style of modular programming, the user can build up a library of callable DSP functions. Differences for particular systems can thus be reduced to data-handling issues rather than the development of new algorithms. While this programming style does not necessarily allow the algorithm to perform its task more quickly, the system designer has more flexibility in establishing how data flows through the system.

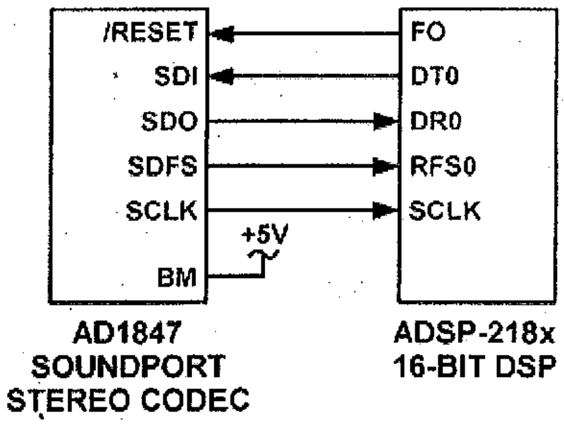
```
#include <def2181.h>        //section A

#define taps 15

#define taps_less_one 14

.section/dm dm_data;        //section B

.var/circ data_buffer[taps]; /* dm data buffer */

.section/pm pm_data;

.var/circ/init24 coefficient[taps] = 'coeff.dat';

.section/pm interrupts;     //section C

start:

jump main; rti; rti;rti;/*0x0000:~reset vector*/

rti; rti; rti; rti;          /* 0x0004:~IRQ2 */

rti; rti; rti; rti;          /* 0x0008:~IRQL1 */

rti; rti; rti; rti;          /* 0x000c:~IRQL0 */

rti; rti; rti; rti;          /* 0x0010:~SPORT0 transmit */

jump fir_start; rti; rti; rti;      /* 0x0014: SPORT0 receive */

rti; rti; rti; rti;          /* 0x0018:~IRQLE */

rti; rti; rti; rti;          /* 0x001c:~BDMA */

rti; rti; rti; rti;          /* 0x0020:~SPORT1 transmit or IRQ1 */

rti; rti; rti; rti;          /* 0x0024:~SPORT0 transmit or IRQ0 */

rti; rti; rti; rti;          /* 0x0028: timer */

rti; rti; rti; rti;          /* 0x002c: power down(non maskable) */
```

```
.section/pm pm_code;        //section D

main:

l0=length (data_buffer);

/* setup circular buffer length */

l4=length(coefficient);     /*setup circular buffer*/

m0=1;                  /* modify=1 for increment */

m4=1;                  /* through buffers */

i0= data_buffer;            /* point to start of buffer */

i4=coefficient;             /* point to start of buffer */

ax0=0;

cntr = length(data_buffer);

/* initialize loop counter */

do clear until ce;

clear : dm(i0,m0) = ax0;    /* clear data buffer */

/* setup divide value for 8 KHz RFS */

ax0 = 0x00c0;               // section E

dm(Sport0_Rfsdiv) = ax0;

/* 1.5385 MHz internal serial clock */

ax0= 0x000c;

dm(Sport0_Sclkdiv) =ax0;

/* multichannel disabled,internally generated sclk, recieve frame sync required,
receive width = 0,transmit frame sync,internal recieve frame sync, u-law
companding, 8-bit words */

ax0=0x69b7;

dm(Sport0_Ctrl_Req) =ax0;

ax0 = 0x1000;               /* enable sport0 */
```

```
dm(Sys_Ctrl_Reg) = ax0;

icntl = 0x00;           /* disable interrupt nesting */

imask = 0x0060;

/* enable sport0 rx and tx interrupts only */

mainloop:

idle;                   /* wait  here for interrupt */

jump mainloop;                /* jump back to idle after rti */

fir_start:

si = rx0;                     /* read from sport0  */

dm(i0,m0) = si;               /* transfer data to buffer */

mr =0,my0 = pm(i4,m4),mx0 = dm(i0,m0);

/*setup multiplier for loop */

cntr = taps_less_one;         /*perform loop taps-1 times */

do convolution until ce;

convolution:

mr = mr + mx0 * my0(ss),my0 = pm(i4,m4),mx0= dm(i0,m0);

/* perform MAC and fetch next values */

mr = mr + mx0 * my0 (rnd);

/*Nth pass of loop with rounding of result */

if mv sat mr;

tx0 = mr1;                    /* write result to sport0 tx */

rti;                    /* return from interrupt */
```

The primary data flows into and out of a digital signal processor can be both parallel and serial. Parallel transfers are typically at least as wide as the native data word of the processor's architecture, i.e.,16 bits for an ADSP-2100 Family processor, 32 bits for the SHARC. Parallel transfers occur via the external memory bus or external host interface bus of the processor. Serial data transfers require considerably fewer interconnections; they are frequently used to communicate with data converters.

Serial Interface: Ease of hardware interfacing is an important element of efficient DSP system implementation. The ADSP-2181 EZ-Kit Lite system uses an AD1847 serial codec (Coder/Decoder). Serial codec permit data transfers via a serial port (SPORT) on the DSP. This serial port is not an RS-232 PC-style asynchronous serial port; it is a 5-wire synchronous interface that passes bit-clock, Receive-data, Transmit-data, and frame-synchronization signals. Major benefits of serial interfaces are low pin count and ease of hardware hookup. The AD1847 requires only 4 signals to interface to the DSP: serial clock, Receive data, Transmit data, and Receive frame-synchronization signal. The serial data stream is time-division multiplexed (TDM), meaning that the same physical line can carry more than one type of information in serial order. In the case of the AD1847 application on EZ-Kit Lite, initiated in the last issue, the serial line carries both left- and right-channel audio information, along with codec control and status information.

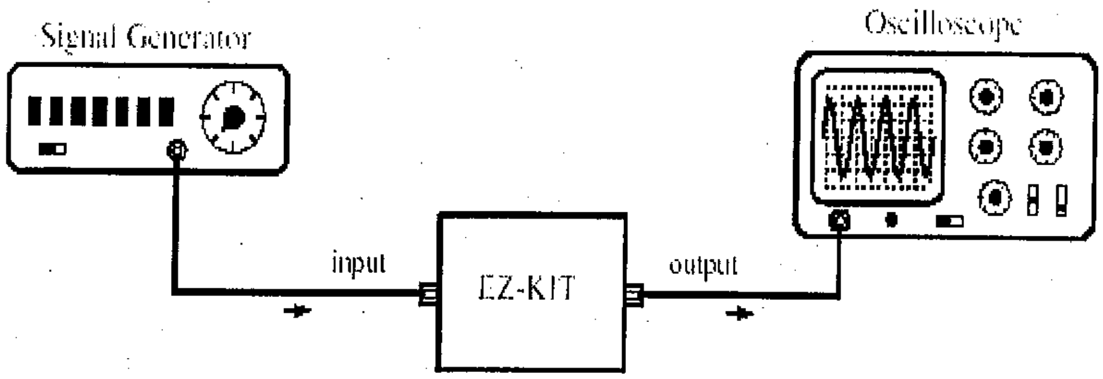


Serial interfacing between digital signal processor and I/O device

As noted earlier, the processor has various means for handling this data. SPORT Interrupts are generated automatically by the serial port hardware for either Receive or Transmit data and for either a single word or a block of words.

5.4 Analog measurements on a DSP system

For just a few moments, forget that you are studying digital techniques. Let's take a look at this from the standpoint of an engineer that specializes in analog electronics. He doesn't care what is inside of the EZ-KIT Lite, only that it has an analog input and an analog output.



Testing the EZ-KIT Lite. Analog engineers test the performance of a system by connecting a signal generator to its input, and an oscilloscope to its output. When a DSP system (such as the EZ-KIT Lite) is tested in this way, it appears to be a virtually perfect analog system

As in figure the traditional analog method of analyzing a "black box," attach a signal generator to the input, and look at the output on an oscilloscope. First, the system is linear (as least as far as this simple test can tell). If a sine wave is placed into the input, a sine wave is observed on the output. If the amplitude or frequency of the input is changed, a corresponding change is seen in the output. When the input frequency is slowly increased, there comes a point where the amplitude of the output sine wave decreases rapidly to zero. That occurs just below one-half the sampling rate, due to the action of the anti-alias filter on the ADC. Now our engineer notices something unknown in the analog world: the system has a perfect linear phase.

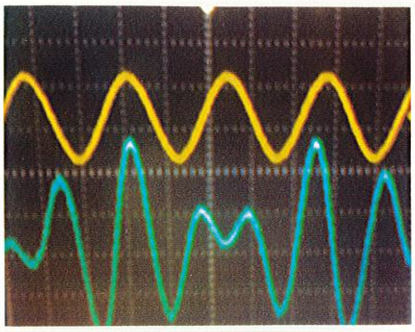


Figure 1. Response of DTMF

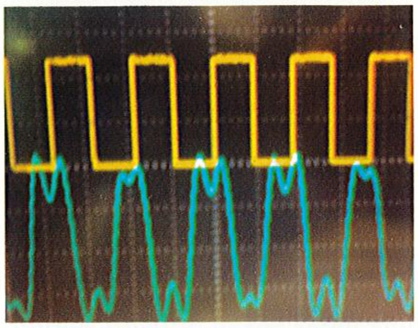


Figure 2 . Response of ADPCM

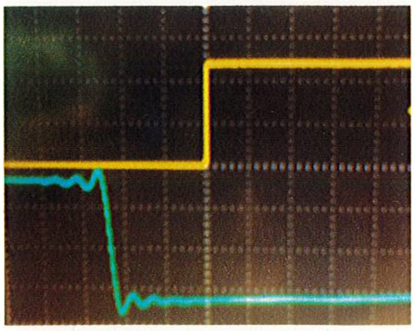


Figure 3. Response of FIR Filter at 130 Hz.

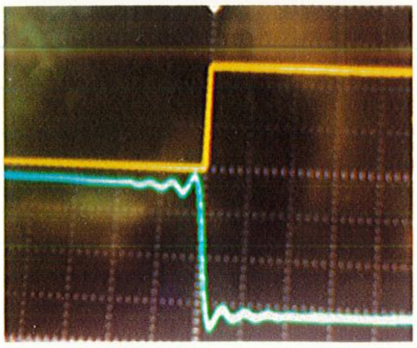


Figure 4. Response of FIR Filter At 140 Hz.

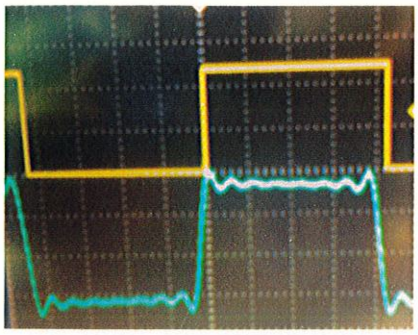


Figure 5. Response of FIR Filter At 150 Hz.

In other words, there is a constant delay between an event occurring in the input signal, and the result of that event in the output signal. Since the center of symmetry is at sample 150, the output signal will be delayed by 150 samples relative to the input signal. If the system is sampling at 8 kHz, for example, this delay will be 18.75 milliseconds. In addition, the sigma-delta converter will also provide a small additional fixed delay.

Conclusion:-

In other words, the device being evaluated is one-hundred times more precise than the measurement tool being used. A proper evaluation of the frequency response would require a specialized instrument, such as a computerized data acquisition system with a 20 bit ADC. It is not surprising that DSPs are often used in measurement instruments to achieve high precision.

Only a decade ago, state-of-the-art signal processing was carried out with precision op amps and similar transistor circuits. Today, the highest quality analog processing is accomplished with digital techniques. It was nice experience to work on ADSP-2181 and enhance our technological knowledge.

Bibliography:

1. A novel approach towards the design of Chebyshev FIR Filter with linear phase Dr. Sunil Bhooshan and Mr. Vinay Kumar.
2. The Scientist and Engineer's Guide to Digital Signal Processing, Second Edition By Steven W. Smith and also DSPguide.com.
3. Oppenheim A.V. and Schafer R.W., Digital Signal Processing.
4. Proakis J.G. and Manolakis D.G., Digital Signal Processing – Principles, Algorithms and Applications.
5. Finite Impulse Response Digital Filters, Vidosav Stojanovisae1,Sinisa Miniae.
6. M. E. Paul, and K. Bruce, C Language Algorithms for Digital Signal Processing, Englewood Cliffs.
7. Sanjit K. Mitra, James F. Kaiser, Hand book for digital signal processing .
8. Samir V. Ginde & Joseph A.N. Noronha, DSP and Filter Design.
9. S. J. Orfanidis, Introduction to Signal Processing, Prentice Hall, Upper Saddle
10. Analog Devices BBS (617) 461-4258, or anonymous ftp site ftp.analog.com, or web site www.analog.com