Digital Signal Processing
18

Digital signal processing (DSP) is one of the great technological innovations of recent times in electronics. This description of DSP for hams was written by Jon Bloom, KE3Z. The basic idea behind DSP is to represent a signal waveform by a sequence of numbers, then process those numbers digitally—usually with a computer—to effect changes to the signal or to extract information from the signal. Similarly, signals can be created by calculating a sequence of numbers that represent the desired waveform.

The advantages of processing signals digitally are: the "circuit" never needs tuning, as the computer program doesn't age or change with temperature variations; flexibility, since the way in which the signal is processed is controlled by software, allowing easy changes to the processing; and some unique capabilities the software approach to signal processing makes available, such as processing that adapts itself to the nature of the incoming signal.

DSP has become of great interest to amateurs in the past few years because the devices needed to do it—fast, dedicated DSP chips—have become easily available and inexpensive. Projects that make use of DSP are showing up more and more often in the amateur magazines and books, and tools that allow the average amateur to work with DSP have become readily obtainable.

In this chapter, we will look at the mechanisms by which DSP is performed, starting with the way waveforms can be turned into sequences of numbers and then, after processing, back into waveforms. We will also look at some of the ways signals can be generated and how they are processed to filter, demodulate and analyze them.

DSP Fundamentals

Any introduction to DSP necessarily begins by covering some fundamental concepts. Among these are the effects of digitizing a waveform to create a sequence of computer words and the effect this operation has on the design of a DSP hardware system.

SAMPLING

The process of generating a sequence of numbers from an analog, or continuous, waveform is known as *sampling*. In sampling a signal, we measure the amplitude, or voltage, of the signal periodically, at a regular interval. This results in a sequence of numbers that state the amplitude of the signal at discrete times. For that reason, DSP is often called *discrete-time* signal processing.

The result of this sampling process might be as shown in **Fig 18.1**. The sequence of numbers tells us the amplitude of the signal at the sampling instant, but we do not know the values of the waveform between the sampling points. You might suspect that we could miss important information about the signal by neglecting the values between the sampling points and, in fact, that can happen if the samples aren't taken close enough together in time. But what is "close enough"? If we sample the sine wave shown in Fig 18.1 one million times per cycle, we are going to end up with a pretty close approximation of the sine wave! But if we sample it once a week, we won't have much useful information. Somewhere between these two extremes lies the minimum sampling rate we can use and still represent the sine wave.

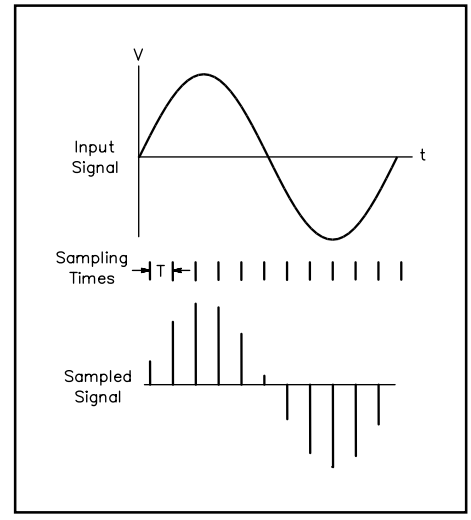


Fig 18.1—Sampling a sine wave. The upper graph shows the input waveform, while the lower graph shows the result of sampling the signal at discrete times.

To discover the minimum acceptable sampling rate, it is useful to look at the sampling process itself. What we put into the sampling circuit is a continuous waveform; what we get out is a set of discrete sample values. One way to look at this process is as a multiplication of two signals, as shown in **Fig 18.2**. One of the signals is the input waveform, while the other is a series of pulses, each with an amplitude of 1 and separated in time by the sampling interval, T. By multiplying these two signals together, we get zero at all times between the sampling pulses and a sample equal to the amplitude of the input waveform at the sampling times.

Now that we can treat sampling as multiplication, we are on more familiar ground. We are used to multiplying signals, since that is precisely what happens in an ideal double-balanced mixer (see the **Mixers** chapter). So we can analyze the result of the multiplication by looking at the frequencies of the two signals, shown in **Fig 18.3**. The input signal is a sine wave in this case, which has a single positive frequency component, f_i, and a negative component, $-f_i$. But what of the sampling pulses? It turns out that the frequency spectrum of the sampling pulses comprises a component at 0 Hz, one at the sampling frequency, F = 1/T, and components at all of the positive and negative harmonics of the sampling frequency: –4F, –3F, –2F, –F, 2F, 3F, 4F and so on. Multiplying—

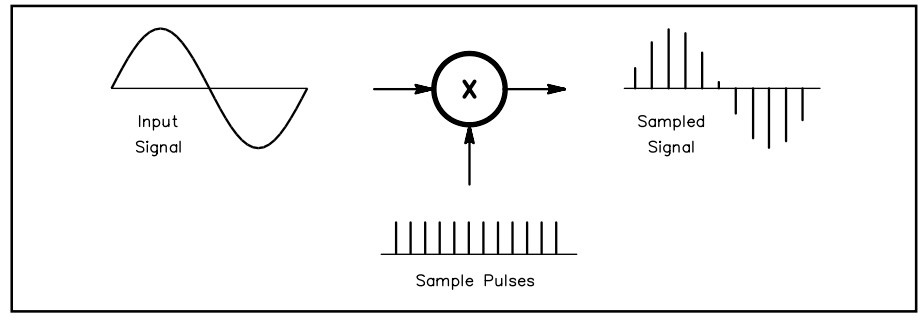


Fig 18.2—Sampling is a multiplication, or mixing, process.

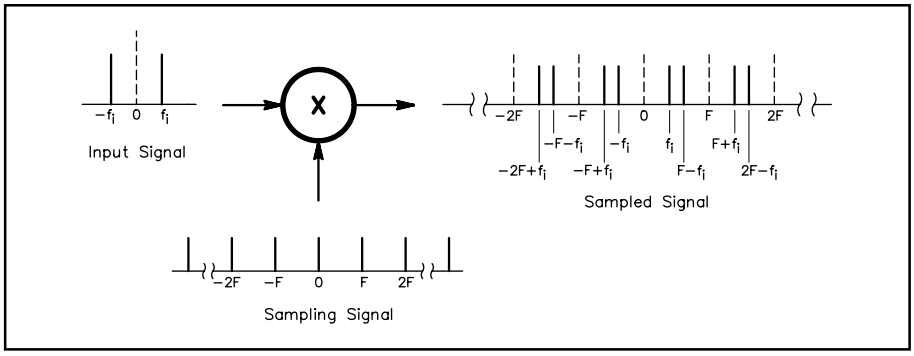


Fig 18.3—The sampled spectrum can be found by mixing the spectrum of the input signal with that of the sample pulses.

mixing—this signal with the input sine wave thus gives us sum and difference frequencies around each of the components of the sampling signal. That is, we have signal components around the sampling frequency, for example, at $F–f_i$ and $F+f_i$. And we have components at $2F±f_i$, at $3F±f_i$, and so on. We also have components around 0 Hz, at f_i and $–f_i$, as well as around the negative-frequency components of the sampling signal, at $–F±f_i$, $–2F±f_i$ and so on.

Now, if our input signal was at a frequency of $F–f_i$, instead of f_i, the input would mix with the 0-Hz component of the sampling signal to give us outputs at $F–f_i$ and $–F+f_i$, with the component at F to give f_i and $2F–f_i$, with the component at –F to give $–f_i$ and $–2F+f_i$, and so on. The result would be indistinguishable from sampling a signal at f_i! Such an input signal is called an alias, since it "looks like" a frequency that it is, in fact, not. Any signal at a frequency above F/2 is indistinguishable from a signal below F/2. That is, a signal at $F/2 + f_i$ is an alias of a signal at $F/2 – f_i$; you can't tell them apart. This fact leads to one of the most fundamental laws of DSP, *the sampling theorem*, which states that you must sample a signal at a sampling rate greater than twice the highest frequency component of the input signal to avoid aliasing.

Normalized Frequencies

The sample values we get from a waveform depend on both the waveform and the sampling rate. If we were to sample a 1-kHz sine wave signal at a 10-kHz rate, we would get the same set of sample values as if we had sampled a 2-kHz signal at a 20-kHz rate. To the computer that processes the numbers, both cases are exactly the same. For that reason, we often find it convenient to *normalize* frequencies in our design and analysis of DSP systems. A normalized frequency is the actual frequency divided by the sampling frequency. Since we should restrict our input signals to actual frequencies of 0 to F/2, this results in normalized frequencies of 0 to 0.5, since (F/2)/F=0.5.

QUANTIZATION

Usually, we perform sampling with an analog-to-digital (A/D) converter. Every so often—at the sampling rate—the processing computer asks the A/D converter for the amplitude of the waveform. Since the A/D converter must give its result as a binary number, it can only respond with one of a limited number of amplitude values. For example, an 8-bit A/D converter can give one of 256 values. When the signal the A/D is measuring falls between two of the values it can represent, the A/D reports the nearest value that it can. This means that, most often, the value reported by the A/D is not exactly the amplitude of the input signal; there is a small error, called the *quantization error*. How large the error is depends on how close the amplitude of the input signal is to one of the binary values the converter can report.

The amplitude relayed by the A/D converter to the computer can, therefore, be thought of as the sum of two signals: the actual input signal and an error "signal." The peak amplitude of the error signal is equal to one-half the amplitude of the least-significant bit (LSB) of the A/D converter, as the difference between the input signal and a reportable binary value cannot exceed this amount. (Of course, this assumes a perfect A/D converter; deficiencies in the A/D can increase this error value.) If the input signal is varying, the error signal will vary as well, as the difference between the actual amplitude and the

reported amplitude changes with each sample. Normally, with a signal composed of numerous frequencies, changing all the time, all possible error values of $\pm^1/_2$ LSB are about equally likely. The result in this case is that the error signal looks random; it is noise—quantization noise. From this, we can calculate an effective A/D signal-to-noise (S/N) ratio, which is the ratio of the desired full-scale signal to the noise term. The result, taking into account the random nature of the error signal, is:

$$S/N = 6.02N + 1.76 \text{ dB} \quad (1)$$

where N is the number of bits of the converter. For example, an 8-bit A/D converter would give an S/N ratio of:

$$6.02(8) + 1.76 = 49.9 \text{ dB}.$$

The amplitude of this noise is distributed across the frequency range of the sampling system. That is, the 49.9 dB S/N ratio of the example converter includes noise from dc to F/2. If digital filtering is used in the processing to reduce the bandwidth, the amount of noise is reduced proportionally. This can be used to advantage, as we will see. But it is important to remember that we *assumed* the error signal was random. If there is a harmonic relationship between the sampling rate and the input signal, the error signal will *not* be random, and the error may show up at discrete frequencies, rather than as random noise.

DSP SYSTEM HARDWARE

The sampling theorem suggests that we had better not allow any signals at frequencies above one-half the sampling rate to get into our A/D converter. If we do, we won't be able to tell whether the signal we're processing is above or below F/2. To eliminate this risk, we usually place a low-pass filter, called an *antialiasing filter* ahead of the A/D. The job of this filter is to attenuate any signal at a frequency above F/2 to the point where its amplitude is negligible. **Fig 18.4** shows the block diagram of a DSP system with an antialiasing filter included.

After we've processed the signal in the computer, we may want to output the processed signal as a waveform. We do this by feeding the samples into a digital-to-analog (D/A) converter. This results in a "staircase" waveform, where the output amplitude is held constant during the sample period.This staircase waveform has a spectrum similar to that of the sampled signal of Fig 18.3, but having the sample amplitudes "connected" by the D/A modifies the spectrum somewhat. Instead of having equal-amplitude components around all harmonics of the sampling frequency, the amplitudes of the components lessen as the frequency is increased. Specifically, the amplitude of any frequency component is:

$$A_o = A_s \frac{\sin \pi f/F}{\pi f/F} \quad (2)$$

where: A_o is the amplitude of frequency f at the output of the D/A converter, A_s is its amplitude at the input of the D/A converter, and F is the sampling frequency. This applies not only to the signals around the sampling frequency and its harmonics, but also to the signals from 0 Hz to F/2. At F/2, the result is:

$$A_o = A_s\ (0.637)$$

which represents a –3.9-dB amplitude error, with less error at lower frequencies, down to no error at 0 Hz. If such an amplitude error is unacceptable in a particular application, you can either choose a faster sampling rate, so that the highest

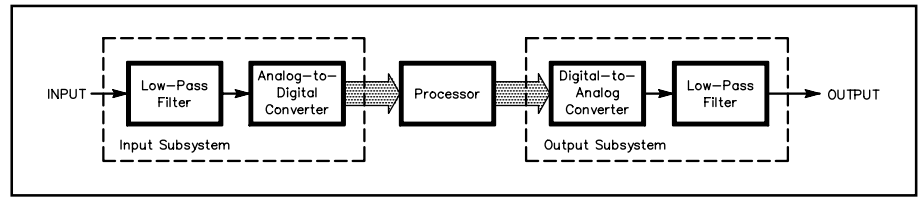


Fig 18.4—Block diagram of a DSP system. The low-pass filter in the input subsystem is the *antialiasing* filter, while the output low-pass filter is the *reconstruction* filter.

signal frequency falls well below F/2 (called *oversampling*), or correct the amplitude error with filtering. Some D/A converters designed for DSP work include internal error correction for this effect.

Most likely, we will want to remove the signal components above F/2 from the output of the D/A converter. We do this by following the D/A converter with a low-pass filter, to remove these components of the signal, as shown in Fig 18.4. Such an output filter is called a *reconstruction filter*.

For both the antialiasing and reconstruction filters, which are analog filters, the necessary filter complexity is determined by how close to F/2 the highest desired signal frequency will be. For example, requiring the filter to pass a signal just below F/2 while rejecting its alias just above F/2 will require a filter with a very steep roll-off. Such filters are both complex and difficult to construct reliably. Also, they typically exhibit a large degree of phase distortion. For that reason, we usually oversample by two to three times. This allows us to relax the requirements for the antialiasing and reconstruction filters, making them simple and benign.

A/D and D/A Converters for DSP

Getting the right kind of A/D converter is important for DSP. A basic A/D converter simply digitizes the voltage that appears at its input. Depending on the type of circuit used in the converter, this digitization may take a relatively short period of time or a long time. How long it takes to perform the conversion affects the highest sampling rate the A/D converter can support. If the converter isn't finished performing its calculations for a given sample, it can't begin calculating the next one. So, one of the key parameters to consider in choosing an A/D converter is its maximum sampling rate.

But there's another issue to consider as well. What happens if the input voltage changes while the A/D converter is doing its processing? The answer is that the A/D will probably report some value between the initial voltage, at the start of the conversion, and the final voltage. This will seriously affect the accuracy of our sample values. Remember, we assumed that each sample value was a snapshot of the input voltage at one instant of time.

The solution to this problem is to add a sample-and-hold (S/H) amplifier ahead of the A/D converter. This circuit is clocked, or strobed, at the same time the A/D converter is commanded to begin its conversion. The S/H amplifier "freezes" its output level, holding a steady voltage on the input of the A/D converter regardless of any succeeding changes in the voltage coming into the S/H amplifier. Often, an S/H amplifier and an A/D converter are packaged into a single integrated circuit part. Such devices are knows as *sampling* A/D converters.

While D/A converters don't have to worry about their inputs changing between samples, they do have to be able to quickly change their output voltages from sample to sample. Be sure to use a D/A converter rated for the output sampling rate you intend to use.

Processing Signal Sequences

While we have to expend a certain amount of effort to turn waveforms into sampled sequences and back into waveforms again, the heart of DSP lies in the processing of those signals. Processing of signals is performed largely by three fundamental operations of DSP: addition, multiplication and delay. Adding two numbers together or multiplying two numbers together are common computer operations, and we won't dwell on them too much. Delay, on the other hand, takes some explaining.

Delaying a signal, in DSP, means processing previous samples of the signal. For example, you might take the current input sample and add its value to that of the previous input sample, or to the sample before that. It's difficult to explain without tiresome mathematics exactly how and why delays enter into our processing so importantly. We can draw an analogy, however, to analog R-L-C circuits, in which the delays (phase shifts) of the inductors and capacitors work together to create frequency-selective circuits. Processing discrete-time signals on the basis of a series of samples performs much the same function as the phase shifts of reactive analog components. Just as the inductor or capacitor stores energy, which is combined with later parts of the applied signal, stored sample values are combined in DSP with later sample values to create similar effects.

We will represent DSP algorithms in two ways: by flow diagrams and by equations. Flow diagrams are made up of the elements of **Fig 18.5**. These provide a convenient way to diagram a DSP algorithm. One item of note is the delay block, labeled z^{-1}. For any given sample time, the output of this block is what was at the input of the block at the previous sample time. Thus the block provides a one-sample delay. It is important to recognize that the signals "step" through the flow diagram. That is, at each sample time, the input sample appears and, at the same time, all of the delay blocks shift their previous inputs to their outputs. Any addition or multiplication takes place (we assume) instantaneously, producing the output. The output then remains stable until the next sample arrives. While real calculations do, of course, require time to complete, the algorithms don't take that into account—and don't need to. Everything happens on the basis of the incoming sequence of sample values.

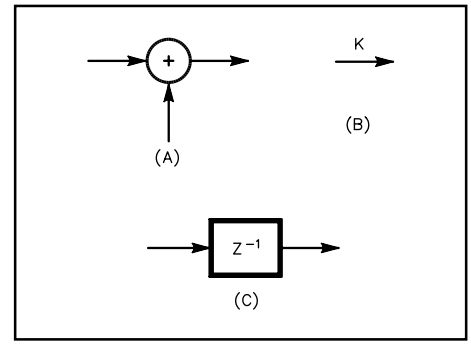


Fig 18.5—Flow diagram symbols. At A, the symbol for adding two sample values. B is the symbol for multiplying a sample value by a constant, K. Delaying the sample value by one sample period is shown at C.

Fig 18.6 shows an example flow diagram. In this simple case, the previous input sample is multiplied by 2 and added to the current input sample. That sum is then added to the previous *output* sample, which is multiplied by –3, to form the current output sample. We have added notation to this diagram to show how the various signals in the diagram are represented mathematically. The key to reading this notation is to understand a term of the form x(n). This can be read as "x as a function of n." The variable n is the *sample index*, an integer value, and sample number n is, in this case, the current input sample. x(n) is simply the amplitude value of the current sample, sample number n. The output of the delay block in the lower left is the previous input sample value. (Recall that the delay block shifts its input to its output each time a new sample arrives.) Thus it is the value of x when n was one less than its present value, or x(n–1). Similarly, y(n) is the current output value, and y(n–1) is the output value at the previous sample time. Putting these signal notations together with the multipliers, or *coefficients*, shown on the diagram lets us construct an equation that describes this algorithm:

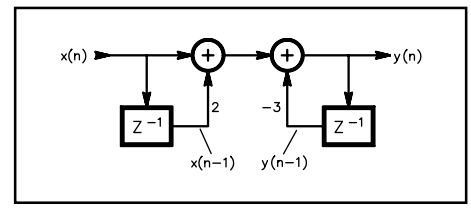


Fig 18.6—An example flow diagram.

$$y(n) = x(n) + 2x(n-1) - 3y(n-1) \quad (3)$$

This equation exactly describes the algorithm diagrammed in Fig 18.6, giving the output sample value for any value of n, based on the current and previous input values and the previous output value. We can use the diagram and the equation interchangeably. Such an equation is called a *difference equation*.

GENERATING SINE WAVES

Up until now, we have been talking about processing a sequence of numbers that came from a sampled waveform. But we also can let the computer calculate a sequence of numbers to generate a signal. One of the easiest—and most useful—signals we can generate in this manner is a sine wave.

One commonly used technique for generating a sine wave is the *phase accumulator* method. We generate our samples at a constant rate—the sampling frequency. For any frequency we wish to generate, we can easily calculate the change in phase of a signal at that frequency between two successive samples. For example, say we are generating samples at a 10-kHz rate—every 0.1 ms. If we want to generate a 1-kHz signal, with a period of 1 ms, we note that the signal changes 36° in 0.1 ms. Therefore, the phase angle of the signal at each sample proceeds:

0°, 36°, 72°, 108°, 144°, 180°, ...

All we need do is find the sine (or cosine, if we prefer) of the current phase angle; that will be the value of our output sample:

sin(0°), sin(36°), sin(72°), sin(108°),...

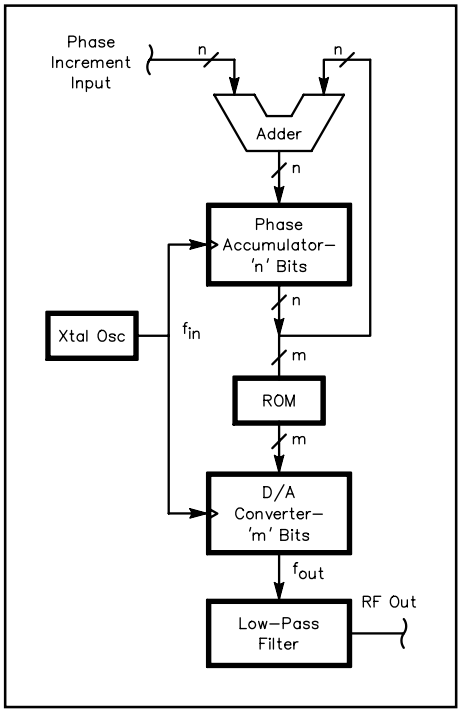


Fig 18.7—Direct digital synthesis (DDS) can be performed using digital hardware, without a DSP chip.

Once the phase passes 360°, it rolls over; it always has a value between 0 and 360°. Finding the sine or cosine can be done in the computer in one of several ways, although most often it is done with a look-up table, as that is the quickest way.

This kind of generator can be implemented directly in digital hardware, as shown in **Fig 18.7**, and is an example of *direct digital synthesis* (DDS).

Another generator is shown in **Fig 18.8**. This flow diagram shows a DSP sine-wave oscillator. Like any oscillator, it has no signal input, just an output. By choosing proper coefficients and placing the correct starting values in the delay elements, we can generate a particular frequency. While this algorithm works well, it suffers from two defects compared to the phase accumulator technique. First, it is difficult to change the frequency while the system is running. You have to change not only the coefficients, but the contents of the storage elements as well. This leads to a phase discontinuity in the output when the change is made,

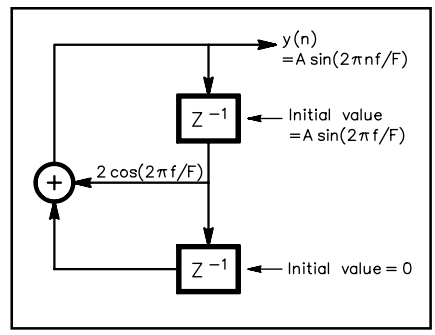


Fig 18.8—A DSP sine-wave oscillator algorithm.

which often is undesirable. The second problem has to do with *finite-length* binary words. Since the coefficient is a number stored in a computer, it must be represented as a set of binary bits. You can't use just any value you want; you have to use values that can be represented by the available number of bits. In this oscillator, the frequency change caused by a one-LSB difference in coefficients is different at low frequencies than at higher frequencies. That's not true of the phase accumulator. So this oscillator is most suitable for applications where a fixed, unchanging frequency is called for.

TIME AND FREQUENCY

Often we are interested in the frequency content of a signal. Using DSP, we have tools that allow us to calculate the frequency content—with some restrictions. The restrictions arise because the frequency content of a signal is not always easy to define. We've all learned that a sine wave consists of a signal at a single frequency, where the frequency is the reciprocal of the period of the sine wave. Actually, that's a simplification.

Consider the signals of **Fig 18.9**. The sine wave at A has a frequency of 1/t, where t is the period of the sine wave. But what about the signal at B? There is a sine wave there, but only one cycle, preceded and followed by a steady zero-volt signal. Since the signal at B is not the same as the signal at A, they *cannot* have the same frequency content! (If they did, they'd be the same signal.) The signal at B is similar to a signal from a CW transmitter keyed on and off, although typically the transmitter would send more than one cycle of the signal at a time. When we abruptly turn a CW transmitter on and off, we get key clicks: signals at frequencies near the frequency of the sine wave. So the turning on and off of the signal changes the frequency content.

What this example demonstrates is that when you analyze the frequency content of an aperiodic signal (one that does not repeat endlessly) over a short period of time, you may get a different result than if you had used a longer period of time. In fact, to be absolutely precise about the frequency content of a real signal, you would have to analyze it over *all* time! That's a bit impractical, of course. Fortunately, if you look at the signal over a relatively long period of time, the difference between what you get as a result and what you would get if you looked at the signal for all of time is pretty small. What's "a relatively long period of time"? That depends on the nature of the signal. In the example of the on-off keyed CW transmitter, you would want to include many of the on-off transitions. The more you include, the more closely your result will come to "reality."

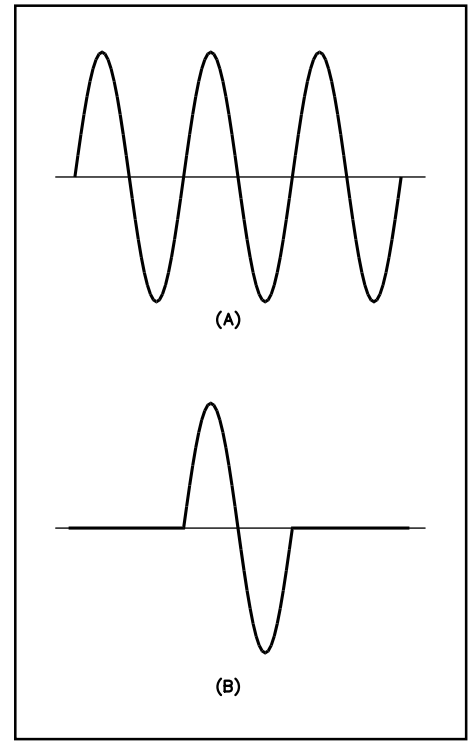


Fig 18.9—These two sine-wave signals have different frequency contents, even though the sine wave in each has the same period.

FOURIER TRANSFORMS

Since we can't look at a signal for all of time, we have to come up with a way of getting close enough. The way we do this is by using a Fourier transform. The Fourier (pronounced **foor**-ee-ay) transform is a mathematical technique for determining the content of a signal. Applied to a signal over a particular period of time, it determines the frequency content of that signal by assuming that the signal being analyzed repeats itself indefinitely.

Of course, when we analyze a real-world signal, such as a couple of seconds of speech, we know that those few seconds of

signal do not, in fact, repeat endlessly. So at best, the Fourier transform can give us only an approximation of the frequency content. But if we look at a large enough period of the signal, that approximation will be pretty good. We also have a mathematical trick up our sleeve that will help us control the error, as we'll see.

The Discrete Fourier Transform (DFT)

In DSP, we make use of a variant of the Fourier transform called the *discrete Fourier transform* (DFT). This is an algorithm that calculates the Fourier transform of a sampled signal. Mathematically, the DFT of a signal is computed thus:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi nk/N} \tag{4}$$

where x(n) is the nth sampled signal input

That $e^{-j2\pi nk/N}$ part may look pretty daunting, but there is a simplification, called Euler's rule, that we can use to state the DFT in more familiar terms:

$$X(k) = \sum_{N=0}^{N-1} x(n)\left[\cos(2\pi nk/N) - j\sin(2\pi nk/N)\right] \tag{5}$$

In this equation, N is the number of samples we're processing, n is the sample index, starting at 0, and x(n) is the value of sample number n. For any value of k, the frequency index, we get X(k), which is the content of the signal at the frequency kF/N, with F being the sampling frequency. We can do this for values of k from 0 to N–1. For example, say we were sampling at 10 kHz and we took 50 samples. Each value of k would represent a frequency of:

$$\frac{k}{N}F = k\frac{10000}{50} = k \bullet 200 \tag{6}$$

so, for k=1, X(k)=X(1) is the content of the signal at 200 Hz. For k=2, X(k)=X(2) is the content of the signal at 400 Hz, and so on. The dc value of the signal is at 0 Hz, so it is given when k=0 and X(k)=X(0).

To calculate X(k) for a particular value of k, we plug k into equation 5, then compute the sum for all of the input sample values. There is a niggling detail left: the value we calculate has both real and imaginary components: it's a complex number. The imaginary component arises because of the *j* in equation 5. What this means is that the signal has both an amplitude and a phase. We can calculate the amplitude and phase from the complex value of X(k) like so:

$$\begin{aligned} X(k) &= a + jb \\ |X(k)| &= \sqrt{a^2 + b^2} \\ \theta(k) &= \tan^{-1}\frac{b}{a} \end{aligned} \tag{7}$$

Here, the values of a and b are what we calculated with cosines and sines in equation 5. |X(k)| is the amplitude, and θ(k) is the phase angle.

Note from equation 6 that if we use values of k greater than N/2, the corresponding frequency for X(k) is greater than F/2. Since the sampling theorem says frequencies above F/2 are aliases, what are these values? It turns out that in the DFT, half of the actual amplitude of a frequency component appears at the expected value of k, and half appears at the alias frequency. If the input samples, x(n), are all real

numbers, the value of X(k) at the alias frequency is the complex conjugate of the value at the actual frequency, meaning the complex number has the same real part and an imaginary part that is equal in value but opposite in sign. Mathematically, we write this as:

$$X(N - k) = X^*(k) \quad (8)$$

What this means in practice is that once we have calculated the value of X(k), we know the value of X(N–k): just reverse the sign of the imaginary part. But even easier, just calculate values of X(k) for k from 0 to (N–1)/2 and then double the calculated amplitude to account for the alias-frequency part. The result is the spectrum of the sampled signal.

There are times, though, when the samples we are processing with the DFT are not real numbers. In that case, the values in the bins where k > N/2 will not be complex conjugates of the bins k < N/2; there will be no simple relationship between a frequency bin and its alias.

Spectral Leakage

In equation 5 we are limited to integer values of k. That means we can calculate, in our example, values of X(k) at 1200 Hz (k = 6) and 1400 Hz (k = 7), but not at 1300 Hz. But what if there *is* a frequency component of 1300 Hz in the signal we are analyzing? Simply, part of that signal shows up in the 1200-Hz "bin," part shows up in the 1400-Hz bin, and smaller parts show up in other nearby bins. This is the error we discussed earlier. It occurs because in our example, a 1300-Hz signal doesn't occur an integer number of times in our 50 samples. That is, if a 1300-Hz sine wave began at the first of the samples, the last of the samples would not occur just as a cycle of the sine wave was completing. Since the DFT assumes the same 50 samples occur over and over, we get a discontinuity at the end of the set of samples. This abrupt discontinuity causes unexpected frequency components, just as does fast on-off keying of a CW transmitter.

This phenomenon is known as *spectral leakage*, since a signal component at a frequency between bins appears to "leak" into adjacent bins. **Fig 18.10** shows an example DFT with input signals of 1000 and 1300 Hz each at the same amplitude. The 1000-Hz signal falls directly on a bin and therefore produces a single line. But for the 1300-Hz signal, it is clear that not only has the signal leaked into nearby bins, but the actual amplitude of the signal isn't obvious, since the signal is divided up among several bins.

We can improve the situation somewhat by taking more samples. Equation 6 shows that increasing N moves the bins closer together. Analyzing a signal that falls between two bins will still cause leakage into nearby bins, but since the bins are closer together the spread in frequency will be less. But this doesn't solve the problem of the amplitude variation.

To minimize that problem, we use a technique known as windowing. (This is the mathematical trick we mentioned earlier.) We multiply each sample of the set we're analyzing by a value determined by the position of that sample in the set. **Fig 18.11** shows a set of samples before and after windowing. The samples near the beginning and end of the sample set have been reduced in amplitude. The effect of this is to reduce the amount of discontinuity that occurs at the end of the sample set and the beginning of the (assumed) identical following set of samples. Reducing this discontinuity reduces the spectral leakage problem.

You don't get something for nothing, however. Obviously, we have distorted the signal we're analyzing. The effect of this shows up in the resulting spectrum, shown in **Fig 18.12**. Now each frequency component is leaked across several frequency bins, even

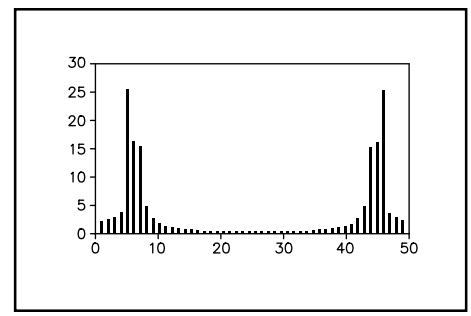


Fig 18.10—The 50-point DFT of a signal with 1000-Hz and 1300-Hz components, sampled at a 20 kHz rate, shows the effect of spectral leakage. The 1000-Hz signal falls exactly on the fifth frequency bin (k = 5) and doesn't leak at all. The 1300-Hz signal falls between bins 6 and 7, causing it to spread over a number of bins.

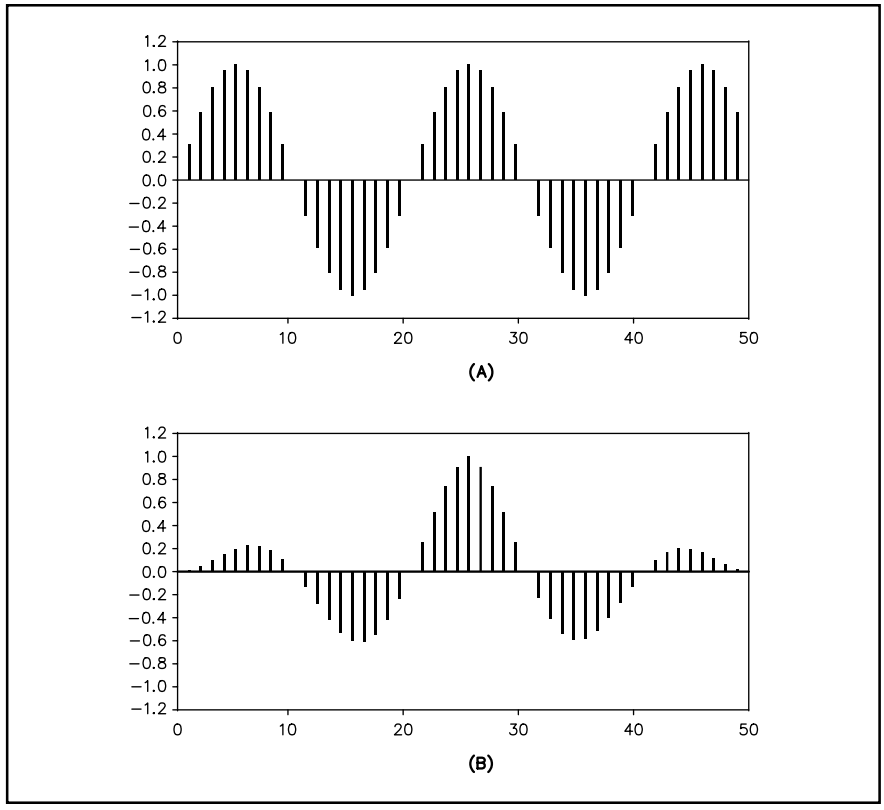


Fig 18.11—Windowing minimizes the effects of discontinuity at the ends of the sample set for the DFT. Here, the samples at A have a window function applied, resulting in the samples at B.

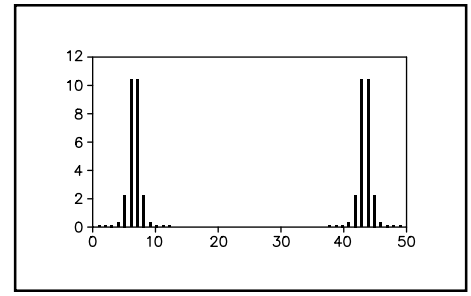


Fig 18.12—The DFT of a 1000-Hz signal, sampled at 20 kHz and windowed. Note that, even though 1000 Hz falls directly on a frequency bin, the signal is spread over several adjacent bins.

if it normally would fall right on a bin. But the leakage is more consistent; you don't get zero leakage at some frequencies and lots of leakage at others. Rather, you get about the same amount at all frequencies. This means that the relative amplitudes of signal components, viewed across several bins of frequency, are nearly the same no matter what the actual frequency of the component. We have traded some resolution for consistent results.

You can't multiply the samples by just any old values to create the windowed set of samples. But a number of window types have been mathematically defined that give the consistent results we are looking for. Among these are the Hamming, Hanning, Blackman and Kaiser windows. Which to use depends in part of how much resolution you are willing to trade for consistency. The more consistent you want the amplitude to be, the less resolution you will get.

The Inverse DFT (IDFT)

Since we now have a way to determine the frequency content of a set of samples, it would be handy to also have a way to relate the frequency content back to the original set of samples. This is done with the *inverse DFT* (IDFT):

$$x(n) = \frac{1}{N} \sum_{n=0}^{N-1} X(k) e^{j 2\pi nk/N} \qquad (9)$$

Except for the fact that now the inputs are the frequency bins, X(k), and the result is a sample value, x(n), this looks very much like equation 4. A 1/N factor has appeared, and the sign of the exponent has changed. Simplifying via Euler's rule gives us:

$$x(n) = \frac{1}{N}\sum_{n=0}^{N-1} X(k)\left[\cos(2\pi nk/N) + j\sin(2\pi nk/N)\right] \quad (10)$$

which looks much like equation 5. So alike are these equations that often the same software routine is used to implement both the DFT and the inverse DFT.

THE FAST FOURIER TRANSFORM (FFT)

Calculating the entire spectrum of a sampled signal, for all values of k from 0 to N–1, requires a lot of calculation. For each value of k, each of the input samples must be processed by taking a sine and cosine, multiplying by the sample value, and adding that to the resulting sum. In our example, where N was 50, we have to do this N(N–1)/2 = 50(50–1)/2 = 1225 times! Even on a fast computer, that's a lot of calculation. And the number of calculations increases by the square of N.

There is help on the way. If we choose a convenient number of samples to analyze, some of the sin(2πnk/N) and cos(2πnk/N) values are the same because the sine and cosine functions are periodic. This allows us to factor out the common sine and cosine values from the DFT sum and combine those repeated multiply-and-add operations into one operation. If we're really clever about selecting the right number of samples, we can do this at a number of places, dramatically reducing the number of calculations we have to do. The result is a fast Fourier transform (FFT) algorithm.

The details of FFT algorithms are beyond the scope of this book, but the result is not: the FFT produces exactly the same results as the DFT, only faster—thus its name. Because it is just a fast DFT, the FFT has the same properties as the DFT, including spectral leakage, so windowing is often used with the FFT as well.

There have been a number of FFT algorithms developed over the years. By far, the most commonly used FFT algorithms are those developed by Blackman and Tukey. These are the *radix-2* algorithms. The convenient number of samples used by these algorithms is a power of two. You can use a radix-2 algorithm on 4 samples, 8 samples, 16 samples, or any number 2^m samples, where m is an integer 2 or greater. The speed improvement of using a radix-2 FFT increases as the number of samples increases, as shown in **Table 18.1**.

THE Z-TRANSFORM

We used equation 3 to mathematically describe the algorithm of Fig 18.6. This equation is useful, but it's hard to manipulate algebraically because it has no common variables: x(n) is a different value from x(n–1). Manipulating the equations that represent DSP systems with algebra is useful because, if we can do it, we can find different ways of implementing the same system, and some algorithms we find will be easier to implement than others. To get the difference equation into a form we can manipulate, we will use the *z-transform*. The mathematics that underlie the z-transform are outside the scope of this book. We will concentrate on the mechanics of using z-transforms.

Earlier, we labeled our delay

Table 18.1
Speed Improvement of the Radix-2 FFT

Number of points (N)	*Number of complex multiplications in DFT = N^2*	*Number of complex multiplications in radix-2 FFT = $(N/2)\log_2 N$*	*Improvement factor*
4	16	4	4.0
8	64	12	5.3
16	256	32	8.0
32	1024	80	12.8
64	4096	192	21.3
128	16384	448	36.6
256	65536	1024	64.0
512	262144	2304	113.8
1024	1048576	5120	204.8

block in the flow diagram with z^{-1}. This is because z^{-1} represents a one-sample delay in a z-transform expression. To convert the difference-equation term x(n–1) to its z-transform, we take the z-transform of x(n), which is X(z), and multiply it by the one-sample delay, z^{-1}. The result is $X(z)z^{-1}$. If the term is x(n–2), we multiply X(z) by two delays: $X(z)z^{-1}z^{-1} = X(z)z^{-2}$. If we perform this operation on all of the terms of equation 3, we get:

$$Y(z) = X(z) + 2X(z)z^{-1} - 3Y(z)z^{-1} \quad (11)$$

Now we can factor out the X(z) and Y(z) terms and solve the equation for Y(z)/X(z), which we denote as H(z):

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1 + 2z^{-1}}{1 + 3z^{-1}} \quad (12)$$

This equation is known as the *transfer function* of the system. It can be expressed in a slightly different, but equivalent, form:

$$H(z) = \frac{1 + 2z^{-1}}{1 + 3z^{-1}} \times \frac{z}{z} = \frac{z + 2}{z + 3} \quad (13)$$

In working with DSP systems, you will often encounter transfer functions like the ones above. To implement the system described by the transfer function, you may have to convert the function back to a difference equation. You do this from a transfer function like that of equation 12 by cross multiplying the two sides, then bring the Y(z) term to one side and all other terms to the other side to get a result like equation 11. Finally, take the *inverse* z-transform of each term, turning, for example, $X(z)z^{-1}$ into x(n–1). From the resulting difference equation, you can either construct a flow diagram or just write your program directly.

IMPULSE RESPONSE

One of the important measures of how a DSP algorithm acts is its *impulse response*. This is the output sequence of a system when the input is a sequence of values equal to 0, followed by a single value equal to 1, followed again by values of zero. This input sequence is called the *unit-impulse sequence*, denoted by δ(n):

$$\delta(n) = \{...,0,0,0,1,0,0,0,...\} \quad (14)$$

Normally, n is zero when the sequence value is 1. The preceding zero values in the sequence are at n = –1, n = –2 and so on, while the following zero values are at n = 1, n = 2 and so on. The output of a system when this sequence is input—its impulse response—is denoted as h(n). For the system of equation 3, the impulse response is:

h(n) = 0 for n<0
h(0) = 1 + 2(0) – 3(0) = 1
h(1) = 0 + 2(1) – 3(1) = –1
h(2) = 0 + 2(0) – 3(–1) = 3
h(4) = 0 + 2(0) – 3(3) = –9
h(5) = 0 + 2(0) – 3(–9) = 27
...

Here, we assume the output was 0 when we started. Note that the output continues to be nonzero indefinitely for n > 0, even though all future inputs are 0. This is an example of an *infinite* impulse response (IIR). If the output had returned to zero and stayed there, it would be a *finite* impulse response (FIR). The infinite nature of the impulse response of this example comes from the feedback of the output

signal into the system. If no feedback exists in the system, the impulse response will be finite.

The usefulness of the impulse response is twofold. First, you can determine exactly how an algorithm will respond to a given input sequence by knowing the algorithm's impulse response. The logic is this: any particular input sample will cause an output that is equal to the impulse response times the input value. If the input value were 1, the output generated by that input would be exactly the impulse response. If the input value were 2, the output values would be doubled. But in a real signal, the input samples are not preceded and followed by an infinite number of zero values; they are preceded and followed by other input sample values. So the output of a system is the sum of the current input sample value times the impulse response, plus the value of the preceding input sample times the impulse response, with that output shifted by one sample time. And the earlier samples contribute shifted, weighted copies of the impulse response as well. The output sequence for a particular input sample, x(n), is:

$$y(n) = x(n)h(0)+x(n-1)h(1)+x(n-2)h(2)+...$$

We can write this in compact form as follows:

$$y(n)=\sum_{k=-\infty}^{\infty} x(n-k)h(k) \qquad (15)$$

Equation 16 is called the *convolution sum*, and the process of taking this sum is called *convolution*. The value of k must be an integer. While k runs from –∞ to +∞, it is only necessary to compute the sum for values of k for which x(n–k)h(k) is not 0. In our example system of equation 3, h(k) is zero for all values of k less than 0. But, since our example impulse response is infinite, we would have to compute the sum for values of k up to the point where h(k) becomes zero, if that ever happens. (It doesn't, in this example.) Some mathematics can be used to show that equation 15 is equivalent to:

$$y(n)=\sum_{k=-\infty}^{\infty} x(k)h(n-k) \qquad (16)$$

We can use these equations interchangeably.

The second useful characteristic of the impulse response is that the frequency response of the algorithm can be obtained by taking the DFT of the impulse response. This is a powerful tool for analyzing a DSP system. Of course, if the impulse response is infinite, you would theoretically need an infinitely long DFT—and an infinite amount of time to calculate it! But in practice, useful IIR systems have impulse responses that approach 0 as k gets large, although they never quite get there. So a very good approximation of the frequency response can be gotten by taking enough of the impulse response sequence that the remaining values are all very close to zero and performing the DFT on that truncated sequence. For an FIR system, on the other hand, an exact frequency response can be obtained by taking the DFT of its impulse response sequence.

The impulse response sequence of an FIR system may be too short for a useful DFT. If, for example, the sequence has only 20 nonzero terms, a 20-point DFT would result in only 10 frequency bins between 0 and F/2. In this case, we simply append zero values to the end of the impulse response sequence to create the desired number of samples, then take the DFT. This approach also allows us to create a sequence length that is usable with an FFT algorithm, speeding our analysis.

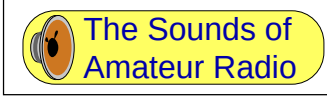
Digital Filters

Filters make up one class of system that is of special interest in DSP. As we have seen, an algorithm that has a particular impulse response also has a particular frequency response, determined by the DFT of the impulse response. So, by creating a system with the proper impulse response we can achieve a particular frequency response—a filter. There are several reasons why DSP filtering might be preferable to using analog filters. The principal reasons are based on the precise, unchanging nature of digital systems. In general, more stringent filter requirements—steeper roll-off in the frequency response, or less distortion in the phase response—call for more complex filters. As a filter gets more complex—adding inductors and capacitors in the case of analog filters, or adding additional delay elements in the digital case—the sensitivity of the filter's response to small errors in the element values becomes more severe. Thus for analog filters, precise values of resistance, inductance and capacitance must be maintained if the filter is to operate as designed. Establishing those precise component values is difficult, and maintaining them during temperature variations and aging of the components is more so. DSP filters, on the other hand, are unchanging. The "component" values consist of numbers stored in a computer, which are not susceptible to temperature changes or aging. For that reason, highly complex filters that would not be viable in the analog realm are easily formed by DSP algorithms.

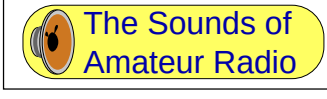
DSP filters can be broadly divided into two classes, depending on whether the impulse response of the filter is finite or infinite. Each class has its advantages and disadvantages; which to use will depend on the requirements of the filter and the system being used.

Designing a filter begins with specification of the desired filter response. The specification must describe the cut-off frequency (or frequencies) of the filter, the allowable amplitude variation in the pass band and the amount of attenuation in the stop band. Refer to the **Filters** chapter for background on filter response specification. In specifying DSP filters, we often use normalized frequencies, since the filter design depends on the ratio of the filter cut-off frequency to the sampling frequency, rather than the actual signal frequency.

Just as the sampled signal includes alias components around the harmonics of the sampling frequency, so too does the frequency response of a DSP filter. It is not possible to use a DSP filter to filter out these alias components. (Unless the sampling rate is changed, as described below.) They exist because of the nature of a discrete-time signal, and a digital filter can't change that. So, a filter that passes a particular frequency also passes all aliases of that frequency.



SSB reception with a DSP filter set for a narrow bandwidth.



SSB reception with a DSP filter wide open.

FIR FILTERS

The basic structure of an FIR (finite-impulse response) filter is shown in **Fig 18.13**. This kind of filter is sometimes referred to as a *transversal* filter. The difference equation of this filter can be determined by inspecting the flow diagram:

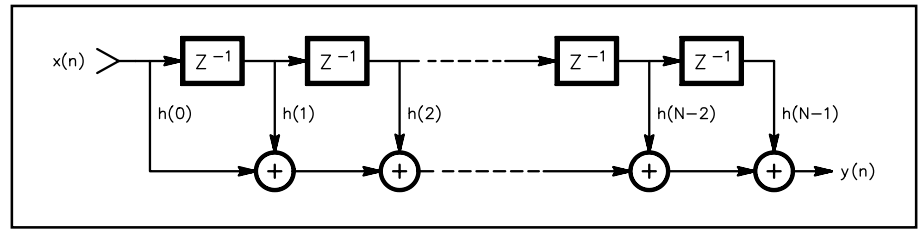


Fig 18.13—Structure of an FIR filter. N is the number of delay elements, or *taps*.

$$y(n) = x(n)\,h(0) + x(n-1)\,h(1) + \ldots + x(n-[N-1])\,h(N-1)$$

Or, writing it in more compact form:

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

Note that this equation is the convolution sum, just like equation 15, except that since h(k) is a finite-length sequence, k runs from 0 to N – 1. Thus the FIR filter directly implements convolution.

The impulse response of the filter of Fig 18.13 is easy to find by feeding the unit-impulse sequence (equation 14) into the filter. The single nonzero input sample first appears at the input, where it is multiplied by the coefficient h(0). Since this sample was preceded by an infinite number of 0 values, all of the delay elements have 0 at their outputs. Only the multiplied input sample contributes to the output; h(0) is the result. When the next sample arrives, the 1 is shifted to the output of the first delay element, to be multiplied by h(1). All other sample values being 0, h(1) is the resulting output. On succeeding samples, as 0 values arrive at the input, the 1 value is shifted successively to each delay-element output, to be multiplied by h(2), h(3), and so on, up to h(N – 1). Thus the impulse response of the filter is simply equal to the coefficients h(0) to h(N – 1).

The trick, of course, is to find the particular impulse response that gives the desired frequency response for the filter. There are two questions: how many filter elements, or *taps*, are needed? And what are the proper coefficient values to use to give the desired response? In general a longer filter—a higher value of N—can provide steeper roll-off in the frequency response. In practice, most FIR filter design approaches start by estimating the number of taps needed, then redesigning the filter if the number of taps selected is found to be too few or too many. Calculating the required coefficients, on the other hand, is more exact, but requires a lot of calculation.

Finding proper coefficient values is complicated by one other issue. We would like the impulse response to be symmetrical about its center. That is, we want h(0) = h(N–1), h(1) = h(N–2), h(2) = h(N–3) and so on. We want this because an FIR filter with a symmetrical impulse response has a constant delay at all frequencies. This constant delay, which can also be stated as a linear phase response, means that the filter will not introduce phase distortion to the signal. For many uses, especially in digital data communication, this is a crucial filter requirement. Not only is the delay of a symmetrical FIR filter constant, it is easily calculated:

$$d = \frac{N-1}{2}T \qquad (17)$$

where d is the filter delay in seconds, N is the number of taps, and T is the sampling interval in seconds.

One feasible approach to designing an FIR filter relies on the facts that the DFT of the impulse response equals the frequency response and that the IDFT can transform a frequency-domain sequence to the time domain. Thus if we take the IDFT of the desired frequency response, we get the needed impulse response. That works, but it suffers from the same discontinuity problem as the DFT itself. In performing the DFT of a signal, this problem shows up as spectral leakage. In the case of finding an impulse response for our FIR filter, it shows up as a ripple in the frequency response. Only at the exact frequencies of the bins of the specified frequency response do we get the correct result; between those bins we get variations in the response. We can attack that problem in the same way we attacked spectral leakage—with windowing. But in doing so, we modify the frequency response, just as we spread out the signal components over several bins in the DFT. That makes it more difficult to get the exact frequency response we want without trial and error.

A better design results from using a complicated design algorithm developed by Parks and McClellan.

This approach results in an *equiripple* design, where all of the passband ripples are of the same amplitude, as are all of the stopband ripples.

Since finding the needed coefficients for a given filter design requires so much calculation, it is a good task for a computer. And DSP filter-design programs are easily available at low cost. For that reason, we will not dwell on the design mathematics; use of a filter-design program to calculate the coefficients is by far the most desirable approach. The Bibliography at the end of this chapter lists some filter-design software tools.

IIR FILTERS

While FIR filters have some exceptionally useful qualities, particularly linear phase response, they require a large number of taps—and a lot of computing power—to implement sharp filters. An IIR filter, on the other hand, can give an equivalent frequency response using fewer calculations. What it will not provide is linear phase response. In circumstances where the computational requirements are of more concern than linear phase response, IIR filters may be desirable.

Unlike the FIR filter, the IIR filter includes feedback—that's what makes its impulse response infinite. Its difference equation shows this:

$$y(n) = A_0x(n) + A_1x(n-1) + \ldots + A_jx(n-j) - B_1y(n-1) - B_2y(n-2) - \ldots - B_ky(n-k) \qquad (18)$$

IIR filters can be implemented using several different algorithms, or structures, as shown in **Fig 18.14**. The structure at A is most easily understood, as it can be drawn directly from inspection of the difference equation.

It may not be obvious, but the structure of Fig 18.14B acts just like the structure in A. It contains fewer delay elements, though, which reduces the amount of storage needed to keep the delayed sample values.

Often, a filter-design program used to design IIR filters will give its result in the form of a transfer function. As explained above, the transfer function can be translated to a difference equation, from which the filter can be drawn or implemented. There is a catch, though. As IIR filters get larger, with larger values of j and k in equation 18, finite-word-length effects become a problem. The coefficients used in the filter cannot be represented exactly in the computer; they can only be approximated by the number of bits used to represent numbers in the machine. While this isn't a big problem for FIR filters, the feedback inherent in IIR filters makes it a concern. Small errors in the coefficients may, after being fed back through a number of delay stages to add to the output many times, produce undesirable effects. To combat this problem, the structure shown in Fig 18.14C may be used. Here, the large filter is broken up into a series of second-order (two delays) filters. Cascading these smaller filters results in the overall response desired. To calculate the coefficients of these smaller filters requires applying some algebra to the transfer function of the larger filter. The desired result is a transfer function of the form:

$$H(z) = \frac{A_{10} + A_{11}z^{-1} + A_{12}z^{-2}}{1 + B_{11}z^{-1} + B_{12}z^{-2}} \times \frac{A_{20} + A_{21}z^{-1} + A_{22}z^{-2}}{1 + B_{21}z^{-1} + B_{22}z^{-2}} \times \ldots \qquad (19)$$

Good filter-design programs will calculate the coefficients of the cascaded sections for you.

The most common design technique used for IIR filters is to design an equivalent analog filter, then transform it to a digital filter. Because of this, many IIR filter-design programs require you to specify the filter shape from among the types of analog filters: Butterworth, Chebyshev or elliptical. See the **Filters** chapter for a description of these filter types.

We assume when analyzing the operation of an IIR system that the delay elements initially store 0 values, until an input signal arrives. Because the impulse response is infinite, initial nonzero values may have long-lasting effects on the output signal. For this reason, any implementation of an IIR filter or system should start by zeroing the storage elements before processing begins.

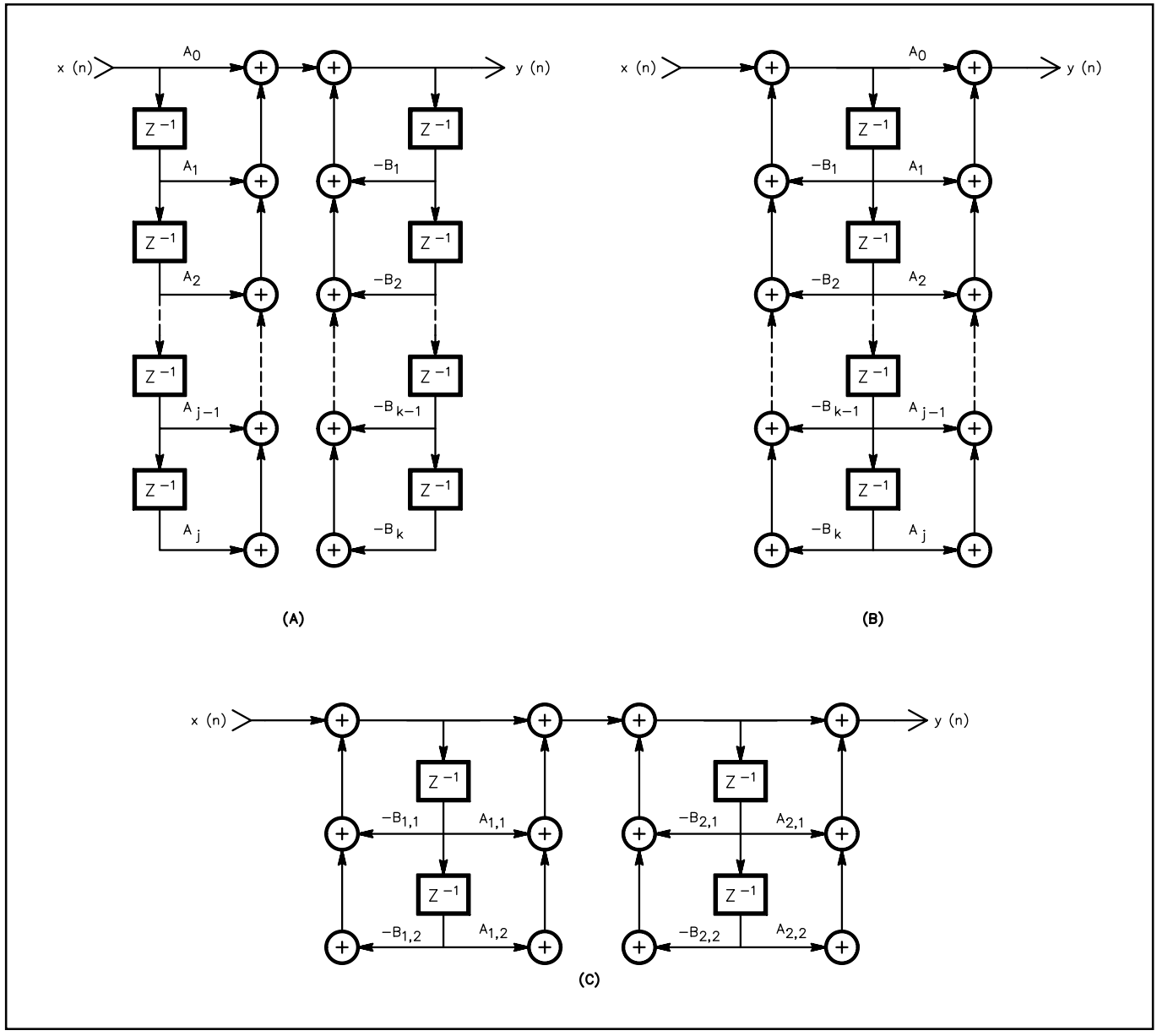


Fig 18.14—IIR filter structures. At A is the direct-form (I) structure; B is the direct-form (II) structure; and C is the cascade structure.

Nonlinear Processes

In analog electronics, we amplify signals, sum them and pass them through frequency-selective circuits. Each of these linear operations has its DSP equivalent, as we've seen. But analog electronics also includes *nonlinear* processing of signals. Examples of nonlinear processes are rectification, clipping, limiting and mixing. One thing that distinguishes nonlinear from linear processing is that in linear systems, the same frequencies that are input to the system appear at the output, possibly with changes in amplitude and/or phase, but without the appearance of frequency components different from those at the input.

As discussed in the **Mixers** chapter, nonlinear processes result in the multiplication of one signal by another (or several others), either explicitly, as in a mixer circuit, or implicitly, as in a nonlinear amplifier or a diode. The same multiplication process is used in DSP, but here we have to be very careful when performing nonlinear operations. A sampled signal comprises not only the frequency of the original signal, but also components around the sampling frequency and its harmonics, as shown in Fig 18.3. Performing a nonlinear operation on sampled signals, therefore, requires that we consider the resulting frequency components and how they will appear in the sampled spectrum.

We begin by taking a closer look at the spectrum of a sampled signal, shown in **Fig 18.15**. Here, the input signal has one frequency component (it's a sine wave), f_1, which shows up in the sampled spectrum at f_1, $-f_1$, $F \pm f_1$, $-F \pm f_1$ and so on. Notice that the spectrum between –F/2 and 0 is exactly like the spectrum from F/2 to F. This is inherent in the sampling process and will always be the case. Because of this, we can concentrate on the spectrum from –F/2 to F/2, knowing that all of the harmonic spectra are simply copies of this spectrum. When we generate new frequency components, by nonlinear operations, that fall above F/2, we can treat them as though they "wrap around" to the negative side, between –F/2 and 0. Generated frequency components more negative than –F/2 wrap around to the positive spectrum.

Now consider what happens when we multiply two sine-wave signals together. This is done by taking each sample of one signal and multiplying it by the corresponding sample of the other signal, as shown in **Fig 18.16**. In analog electronics, we learned through trigonometry that multiplying two sine waves produces sum and difference frequencies. That's true here, too, but we'll look at it in a different, equally valid, way, shown in **Fig 18.17**. We will consider that the positive frequency component of one signal, f_2, shifts the other signal, f_1—both its positive and negative frequency components—up in frequency by the frequency value of f_2. Similarly, the negative frequency

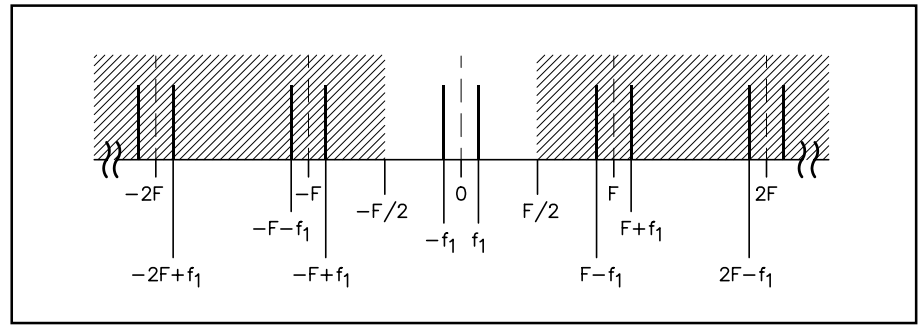


Fig 18.15—The spectrum of a sampled signal can be analyzed by referring to only the part from –F/2 to F/2 (F is the sampling frequency), the unshaded part of this diagram. The spectrum around each harmonic of the sampling frequency is a copy of the unshaded spectrum.

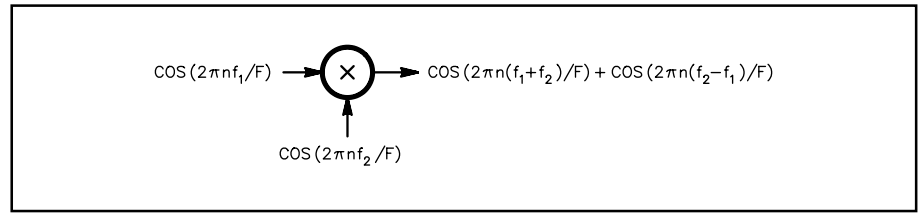


Fig 18.16—Mixing two sine waves is the same as multiplying them. For real-number signals, this results in two signals, the sum and difference of the input signals.

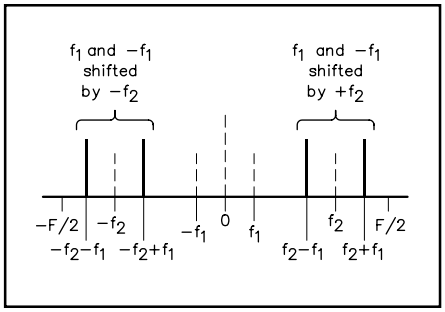


Fig 18.17—The mixing process can be thought of as shifting one signal's frequency components up by the positive frequency of the other signal and down by the negative frequency of the other signal.

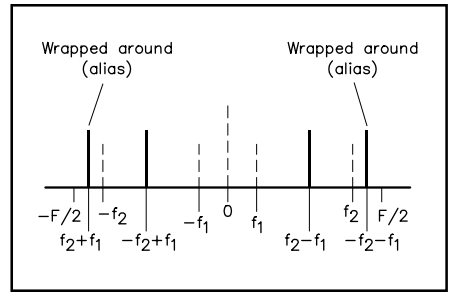


Fig 18.18—If the frequency shift caused by mixing causes a component to exceed the F/2 boundary, the signal will wrap to the opposite end of the spectrum, as shown here.

component of f_2 shifts the components of f_1 down by the same amount. The result is four frequency components, two positive and two negative, as shown in Fig 18.17. We could have chosen f_2 as our shifted signal and f_1 as the amount to shift by and gotten the same resulting spectrum.

Fig 18.18 shows the result if this process ends up shifting a signal beyond the F/2 limit, wrapping it around to the other side of the spectrum. Note that, in this case, the wrapped components appear at frequencies different from where they would be if we had done the mixing with analog electronics. This is because of aliasing, which occurs when a frequency component exceeds F/2.

An example shows why analyzing the effect of a nonlinear operation in this way is important. We can simulate a half-wave rectifier by replacing all of the negative-amplitude samples of a signal with zeros. We might be tempted to do that to, for example, demodulate an AM signal. But a look at the spectral result will show this to be a poor technique. Mathematically, half-wave rectification of a sine wave is like multiplying the input signal by a square wave that has the same frequency as the input signal and amplitude values of +1 and 0. Spectrally, the square wave comprises a dc component and frequency components at the fundamental and all of the odd harmonics of the fundamental. So, the result of rectification is to create frequency-shifted copies of the input signal around 0 Hz and around the odd harmonics of the square wave. Some of these harmonics will show up at frequencies above F/2, no matter what F we choose, and these components will alias into the spectrum below F/2. The result will be nothing like what we get with a physical half-wave rectifier, and most likely nothing like what we wanted. The lesson is that we have to be very careful that our nonlinear operations do not generate unwanted frequency components that show up as aliases. That doesn't mean we can't *do* nonlinear operations, just that we need to exercise caution.

COMPLEX AND ANALYTIC SIGNALS

As we've seen, multiplying two signals shifts the positive and negative components of one of the signals in two directions, generating two sets of frequency components. In analog electronics, we deal with this reality by using filters to eliminate the unwanted second component, leaving only the desirable one. We can do that in DSP, too, but there is another way that is often better.

In all of our preceding discussion, each positive frequency component was mirrored by a corresponding negative frequency component. This is a characteristic of any signal that is composed of amplitude values that are only real numbers. But if we can create a signal that is composed of *complex* amplitude values, this need not be the case. In fact, a complex signal can have only positive-frequency components or only negative-frequency components. Such a signal is called an *analytic* signal.

Consider the usefulness of such signals. If we multiply together two single-frequency signals that have only positive-frequency components, the resulting spectrum is simply a frequency component at the sum of the frequencies of the two signals; there are no negative frequencies present to get shifted into the positive frequency range. This gives us a pure frequency shift, rather than the sum-and-difference result of multiplying two real-value signals.

A sampled, single-frequency analytic signal has the form:

$$x(n) = A\cos(2\pi nf/F) + jA\sin(2\pi nf/F) \quad (20)$$

where A is the peak amplitude of the sine wave, f is the frequency of the signal and F is the sampling frequency. This signal has only positive frequencies. A signal of the form:

$$x(n) = A\cos(2\pi nf/F) - jA\sin(2\pi nf/F) \quad (21)$$

has only negative frequencies. An analytic signal that comprises multiple positive-frequency components is made up of a sum of components of the form of equation 20. That means that the imaginary part of the signal is equal to the real part, but shifted 90° at all frequencies.

In a computer, such as a DSP system, we handle complex numbers by operating on the real and imaginary parts separately. We call the real part of the analytic signal the I (in-phase) component and the imaginary part the Q (quadrature) component. Complex arithmetic dictates that, when we add two complex values, we add the real parts together then we add the imaginary parts together; we still keep the real result separate from the imaginary result. Complex multiplication is a bit more involved. We can multiply two complex numbers like so:

$$(a + jb)(c + jd) = (ac - bd) + (ad+bc) \quad (22)$$

It is easy to generate a single-frequency analytic signal like that of equation 20. Referring to the section on Generating Sine Waves, we can use the phase-accumulator method to generate the I component of the signal, then subtract 90° from the current phase angle and compute the output value for that angle to get the Q component.

There also is an oscillator structure, shown in **Fig 18.19**, we can use to generate the I and Q components for a single-frequency complex signal.

HILBERT TRANSFORMERS

But what if we want to create an analytic signal from a sampled, real signal? We need to shift all of the frequency components of the sampled signal by 90°. Fortunately, in DSP we have a straightforward way to do that: the *Hilbert transformer*. Recall that in the FIR Filters section we noted that an FIR filter with a symmetrical impulse response exhibits a constant delay of (N–1)/2 sample periods. It turns out that an FIR filter with an *anti*symmetrical impulse response—that is, h(0) = –h(N – 1), h(1) = –h(N – 2), and so on—produces a delay of (N – 1)/2 *and a shift of 90°* at all frequencies. This is exactly the kind of filter we need to create the Q component of our analytic signal!

A system using a Hilbert transformer to create an analytic signal is shown in **Fig 18.20**. Since the Hilbert transformer includes not only a 90° phase shift, but also a fixed delay of (N – 1)/2 sample periods, we need an (N – 1)/2 delay in the I channel so that the difference between the two channels becomes solely the 90° phase shift. We also need to have the amplitudes of the two channels the same, so the I channel should have the same

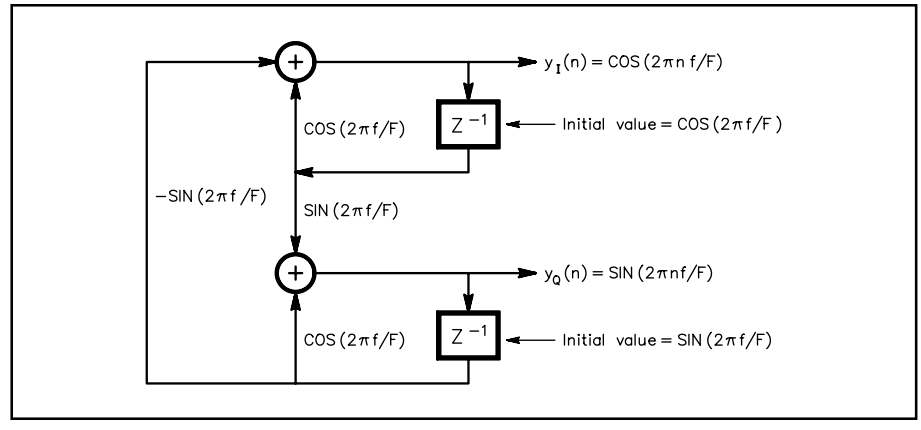


Fig 18.19—A quadrature sine-wave oscillator provides two sine waves, with a 90° phase difference between them.

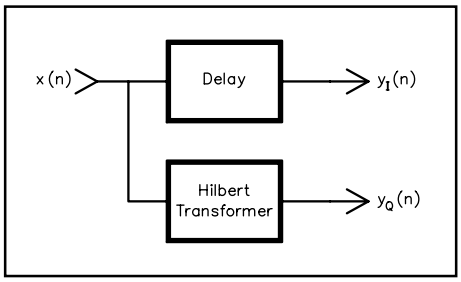


Fig 18.20—Generating an analytic signal from a sampled input signal requires passing the input through a Hilbert transformer filter to generate the Q (quadrature) channel. The I (in-phase) channel is created by passing the signal through a delay equal to the fixed delay of the Q channel.

frequency response as the Hilbert transformer in the Q channel. A Hilbert transformer filter can be designed by most FIR filter-design programs.

When you use analytic signals to perform frequency shifting, you may at some point end up with a signal you want to output to the D/A converter. The D/A, of course, handles only real numbers. Feeding just the real part of the analytic signal into the D/A produces an output waveform that has both positive and negative components, but that's what we expect of a real signal. So, once the processing of the analytic signal is complete, we simply discard the Q-channel signal and use the I-channel values for our output. This may allow us to skip computing the output Q-channel values altogether, as shown in **Fig 18.21**. Note, though, that we do need the Q channel to compute the real part of the frequency-shifted signal, because the multiplied real values include terms from the imaginary part, as shown in equation 22.

A complex signal can also have positive and negative frequencies that are not mirror images of one another. One way of generating such a signal is shown in **Fig 18.22**. Here, a real input signal is formed into a complex signal and frequency shifted, both at the same time. (This is an example of a *half-complex mixer*. It takes in a real signal and produces a complex signal.) With a complex signal such as this, we cannot simply output the real part and expect only the positive frequencies to generate mirror images; the negative frequencies will have mirror images in the positive part of the spectrum, too. Still, such complex signals can be useful at intermediate steps in the processing, before a real signal is output.

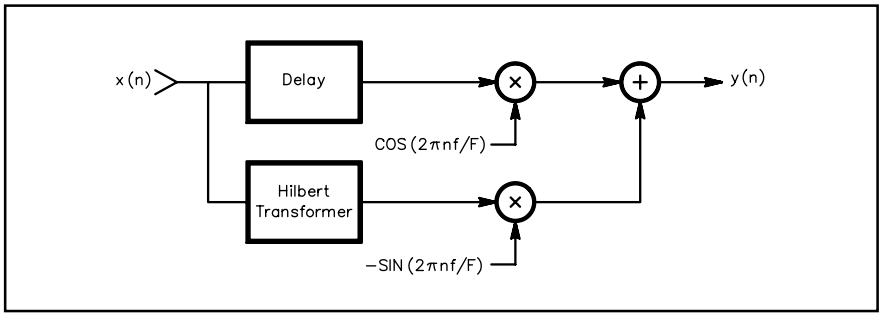


Fig 18.21—Frequency shifting a signal to produce a real-number result doesn't require calculation of the imaginary part of the output, but the imaginary part of the analytic signal does play a part in finding the real part of the output.

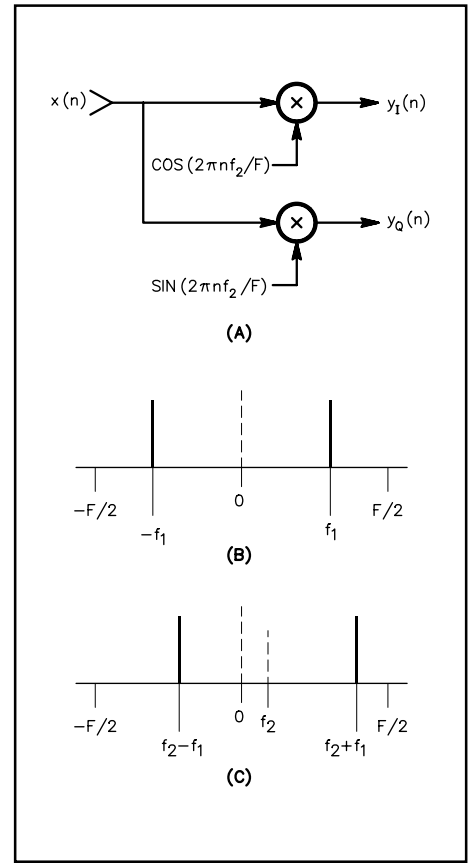


Fig 18.22—Generating a frequency-shifted complex signal from a real signal can be accomplished by multiplying the real signal by a single-frequency analytic signal, as shown at A. Note that the resulting spectrum, shown at B, has both positive and negative frequency components, but they are not mirror images of one another.

Demodulating Signals

One important application of DSP to Amateur Radio is in the demodulation of signals. Although DSP hardware isn't quite yet up to the task of processing directly at amateur operating frequencies, it can demodulate signals at audio and low-IF. An application of audio-frequency demodulation is in the field of modems for digital communication. These devices modulate an audio signal to send digital data and demodulate the received signal to recover the digital data. We also are seeing DSP used more and more as the principal demodulation means in communication receivers, where the DSP operates at a low IF that is mixed from the operating frequency by analog electronics.

There are many algorithms you can use to demodulate each type of modulated signal. We present here a sample of some of the more commonly used techniques. We begin by generating quadrature signals at *baseband* (centered on 0 Hz). From these, we will detect the modulating waveform. The incoming signal consists of a carrier frequency that is modulated in some way. To generate our quadrature baseband signals, we multiply the incoming signal by two signals that are each at the carrier frequency but 90° different in phase. This scheme, shown in **Fig 18.23**, is similar to the half-complex mixer described above. It shifts the signal so that it is centered at 0 Hz in both the I and Q channels. A low-pass filter removes the unwanted negative-frequency components, along with, possibly, filtering out signals that lie outside the bandwidth of the desired signal. The I and Q channel sequences can then be used to demodulate the signal.

AM DEMODULATION

One's first inclination is to demodulate an AM signal by rectification of the signal. But, as explained in the Nonlinear Processes section, that's a technique fraught with peril. A better way is to use the I and Q channels developed in Fig 18.23. These two signals together describe (mathematically) the incoming signal as a rotating vector, with the I channel holding the x-axis (real) component of the vector and the Q channel holding the y-axis (imaginary) part. (See **Fig 18.24**.) All we need to do is to find the length of the vector for each sample of the signal; the vector length is the amplitude of the signal, which is what we want to detect.

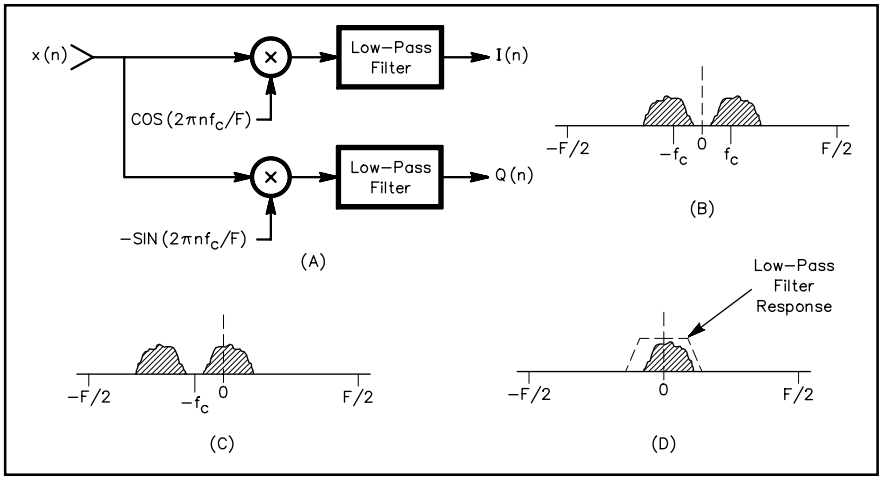


Fig 18.23—Demodulation begins by generating baseband I and Q channels. The technique is shown at A, where an analytic signal at the carrier frequency, $-f_c$, multiplies the input signal, whose spectrum is shown at B. The result of this multiplication is the spectrum at C. This signal is low-pass filtered to remove the unwanted alias spectrum and, possibly, unwanted signals near the desired passband. After filtering, the spectrum at D results.

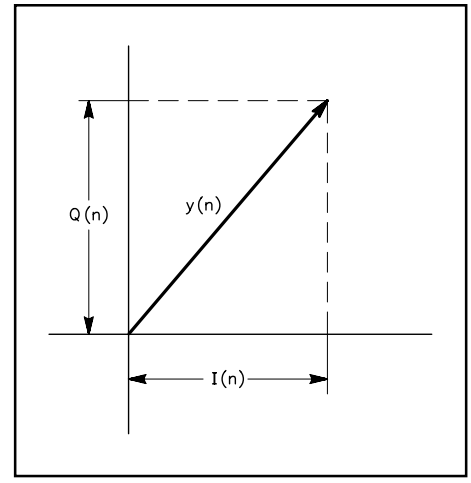


Fig 18.24—The I and Q channels together describe the signal as a rotating vector. From the I and Q sample values we can compute the instantaneous amplitude or phase of the signal.

Finding the length of the vector can be done using the Pythagorean theorem:

$$|y(n)| = \sqrt{[I(n)]^2 + [Q(n)]^2} \qquad (23)$$

For each sample, then, we calculate equation 23, using the current I- and Q-channel sample values. We can filter y(n) to remove the dc component and feed the result to our output D/A converter to produce the detected audio.

Since DSP chips, and computers in general, don't usually provide a square-root function, the program to implement AM detection will have to calculate the square root. This is most speedily done using a look-up table, although a square-root algorithm can be used if desired.

FM DEMODULATION

FM detection is a bit trickier than AM detection. The method described here begins by detecting not the frequency, but the phase of the incoming signal. Since the I and Q channels describe the incoming signal as a vector, we can find the phase of the signal by finding the angle of the vector described by the I and Q signals. This is done using trigonometry:

$$y_p(n) = \tan^{-1}\left[\frac{Q(n)}{I(n)}\right] \qquad (24)$$

Once again, we will most likely use a look-up table to find the arc tangent.

This scheme performs phase detection of the signal, not frequency detection. We can convert the output to a demodulated FM signal by passing the result of equation 24 through a *differentiator* filter. FIR filter design programs can usually generate a design for a differentiator filter. The resulting system is shown in **Fig 18.25**.

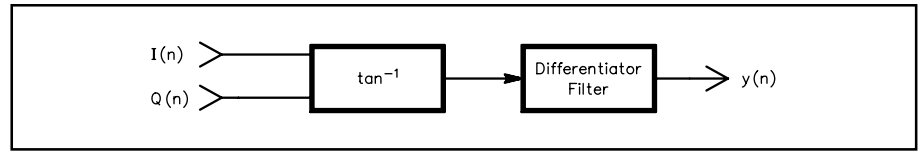


Fig 18.25—FM demodulation is performed by phase detecting the signal via the arc tangent function, then passing the result through a differentiator.

SSB DEMODULATION

The technique we will use for SSB demodulation has long been known in analog electronics. Called the *phasing method*, it is shown in **Fig 18.26**. The Q-channel signal is passed through a Hilbert transformer FIR filter to further shift it by 90°. The I channel is delayed

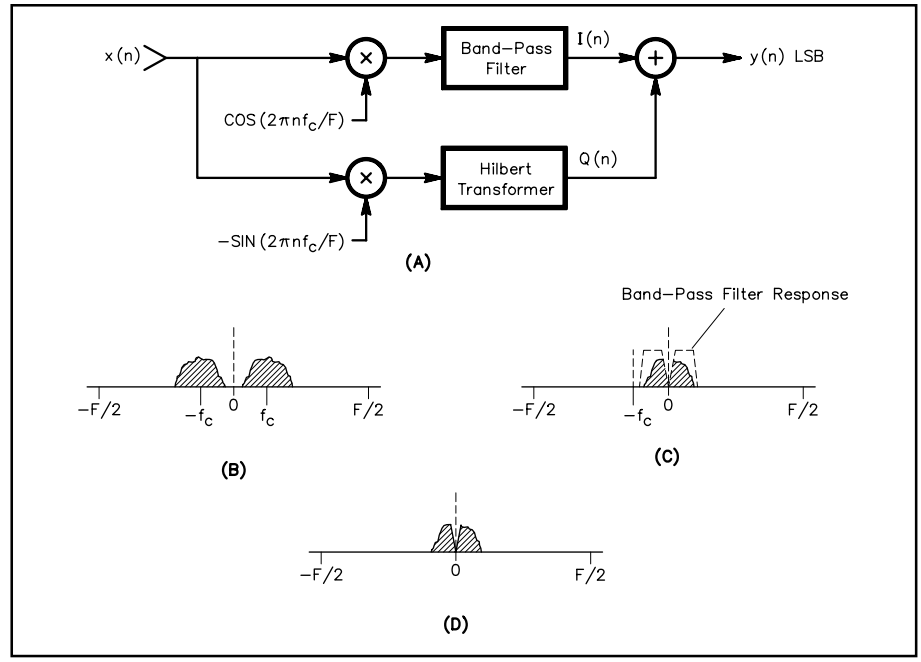


Fig 18.26—The phasing method of SSB demodulation, as implemented in DSP. The system is shown at A. The input signal (B) is mixed with an analytic signal at the carrier frequency, $-f_c$, then the Q channel is phase-shifted via a Hilbert transform, while the I channel is band-pass filtered with a delay equal to that of the Q-channel filter, producing the spectrum at C. Summing the two channels gives the spectrum at D, which represents the lower sideband of the original signal.

by an amount equal to the fixed delay of the Q-channel filter, (N – 1)/2 samples. If an odd number of taps is used in the Q-channel filter, the needed delay of the I channel will be an integral number of samples. The delayed I channel and the transformed Q channel are then summed.

We originally generated our I and Q channels by mixing the incoming signal with a signal at the carrier frequency. That places the lower sidebands below 0 Hz and the upper sidebands above 0 Hz. Summing the two channels causes the positive frequencies in the signal to sum to zero, while the negative frequencies add. Since the result of the summation is a real signal, these resulting frequencies are mirrored in the positive part of the spectrum. What we have done is eliminate the part of the signal above 0 Hz—the upper sideband part—while preserving the part of the signal below 0 Hz—the lower sideband part. If instead of adding the two channels we subtract them, we preserve the upper sideband part and eliminate the lower sideband part.

Decimation and Interpolation

It is often useful to change the effective sampling rate of an existing sampled signal. For example, say you have a system sampling at a 21-kHz rate and you want to filter a 600-Hz signal with a 100-Hz-wide band-pass filter. You could design a filter to do that directly, but it would likely be a very complex filter requiring a lot of processor power to implement. The filter would be easier if the sampling rate were lower, say 7 kHz, since the normalized filter width would be wider. (A 100-Hz-wide filter for a 21-kHz signal would have a normalized width of 100/21000=0.0048, while if the sampling rate were 7000 Hz the normalized width would be 100/7000=0.014.) You may not be able to change the *actual* sampling rate—perhaps the available antialiasing filter won't allow sampling at a lower rate—but you can change the *effective* sampling rate by processing.

DECIMATION

The reduction of the sampling rate by processing is known as *decimation*. The procedure is simple: just throw away the unwanted samples. For example, to reduce the effective sampling rate from 21-kHz to 7 kHz, throw away 2 out of 3 of the incoming samples. This procedure allows you to divide the sampling rate by any integer value.

Of course, throwing away samples changes the signal being processed. **Fig 18.27** shows the effect of decimating a signal by a factor of 3 by keeping only every third sample. F_1 is the original sampling rate, and F_2 is the new sampling rate, $^1/_3$ the original. The resulting signal is indistinguishable from a signal that was sampled at $^1/_3$ the original sampling rate. This means that it contains alias components around the harmonics of F_2. More importantly, it means that any signals present in the original sampled signal at frequencies above $F_2/2$ may alias into the range 0 to $F_2/2$, just as they would have if the signal had actually been sampled at F_2. To eliminate this possibility, it is necessary to digitally filter out any such signals *before* performing the decimation. This can be done with a low-pass filter, at the original sampling rate, that cuts off at $F_2/2$. This filter is called a *decimation filter*.

It may seem like this is no improvement to our example system; now we have to have two filters: a decimation filter and our 600-Hz band-pass filter. But the combined processing of these two filters is less than the processing we would need for the single filter at the original sampling rate. **Fig 18.28** shows why this is so. The decimation filter needs only to attenuate those signals that would alias into the 100-Hz passband of the final 600-Hz filter. Signals that alias into the frequency range above that filter and below $F_2/2$

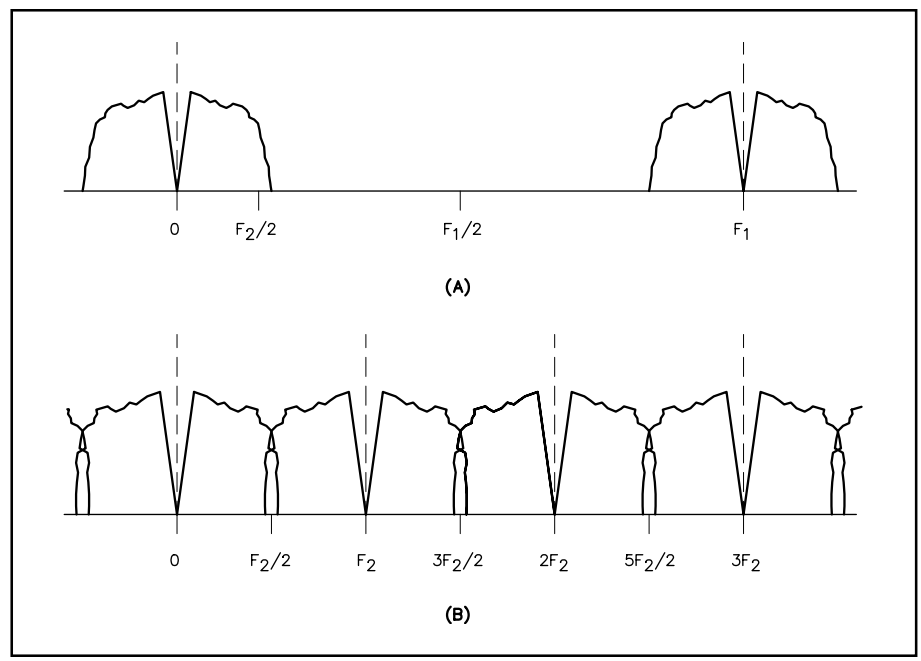


Fig 18.27—The signal at A is sampled at an F_1 rate. At B, the signal has been decimated by a factor of 3 by throwing away two out of three samples. The result is that alias components have been formed around harmonics of the new sampling rate, F_2. Note that in the original spectrum, signal components existed at frequencies above $F_2/2$. These components alias into the range 0 to $F_2/2$ after decimation.

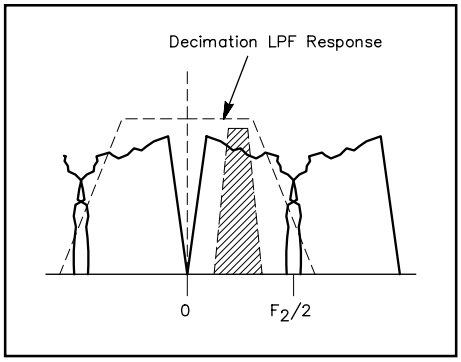


Fig 18.28—In this example, the decimation low-pass filter has to pass the frequencies that will exist after the final filter, shown as a shaded area, while eliminating frequencies that might alias into the final filter.

will be removed by the band-pass filter. That means that the decimation filter need not have a particularly sharp cutoff, so it doesn't have to be a complex filter, costly in terms of processing requirements.

INTERPOLATION

Just as we sometimes want to decimate a signal to reduce its effective sampling rate, we also sometimes want to *interpolate* a signal to increase its rate. Referring to our example of decimation, we may want to output the filtered 600-Hz signal, sampled at 7 kHz, through a system that has a reconstruction filter that cuts off well above 3500 Hz (half the sampling frequency). We can do this by increasing the effective sampling rate to one that accommodates our reconstruction filter.

Just as decimation was performed by removing samples, interpolation is performed by inserting them. If we want to raise our 7-kHz sampling rate by a factor of 3, to 21 kHz, we need to have three times as many samples. We can do that by adding two new samples between each of the existing samples. Usually, we add samples whose value is zero. While this increases the number of samples, it does not change the content of the signal. Specifically, the alias components that lie on either side of the old 7-kHz sampling frequency and its harmonics are still present. To make use of the new signal, we need to digitally filter out all of these components except those around 600 Hz. So, we need a low-pass filter, operating at the sampling frequency of 21 kHz, to eliminate these unwanted signals so they won't appear in the output.

In this example, we know that, because of our 100-Hz-wide filter, all of the signal appears in narrow bands centered on 600 Hz, 7000–600=6400 Hz, 7000+600=7600 Hz, and so on. The highest frequency we need to pass through our interpolation low-pass filter is 650 Hz. The lowest frequency we need to reject is 6350 Hz. We can design our low-pass filter accordingly. With this much difference between our passband and stopband frequencies, we'll find that the needed interpolation filter is simple and won't take much processing.

OVERSAMPLING

One place where decimation and interpolation are often used is to implement oversampling—sampling at a rate much higher than the sampling theorem demands. One reason to use oversampling was shown above: to relax the requirements of the antialiasing and reconstruction low-pass filters. Another advantage of oversampling is in noise reduction. As explained above, if the quantization noise that arises from quantizing the input signal is random in nature, it will be distributed evenly throughout the spectrum. If we make use of oversampling, the noise of interest will be distributed evenly from 0 Hz up to one half the sampling frequency. When we pass the digitized signal through our decimation low-pass filter, much of this noise will be filtered out. This increases the effective signal-to-noise ratio, since the signal is unchanged but the total noise is reduced. This approach can allow use of a low-resolution, but fast, A/D converter to act as if it had more resolution—more bits.

We stress again that this technique assumes randomness on the part of the quantization noise. That may not always be the case. In situations where that can't be assumed by the characteristics of the input signal, *dithering* is sometimes used. Dithering is the introduction of noise into the input signal, before digitization. Usually, the amplitude of this noise is equal to several quantization levels—several times the A/D LSB value. Adding noise may seem to defeat the purpose of

oversampling, but the trick is to add noise that is limited in frequency to a range that will fall outside the passband of the decimation filter. That way, the noise doesn't contribute to the final signal-to-noise ratio but does force the quantization noise to become random. Such a scheme is shown in **Fig 18.29**.

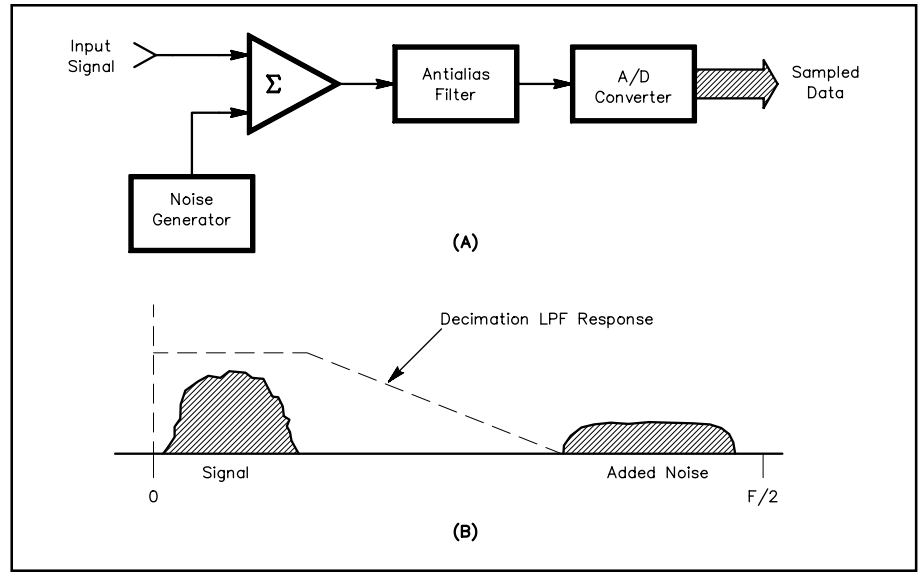


Fig 18.29—Dithering is accomplished by adding noise to an oversampled signal. The noise should fall outside the passband of the digital decimation filter.

DSP Hardware and Development Tools

DSP relies on operations—addition, multiplication and shifting—that are common computer operations. But the large number of such operations needed by any useful DSP algorithm, and the small amount of time available to do them—the interval between two incoming samples—means that general-purpose processors find it difficult to process signals even at audio frequencies. For that reason, most real-time DSP is performed by specialized processors.

DSP CHIPS

Processors for DSP differ from general-purpose processors in important ways. The most important differences exist to optimize the repeated multiply-add-shift operation of DSP algorithms. One of these optimizations is the use of the *Harvard architecture*. This scheme of computer organization has separate program memory and data memory. The program instructions and constant values are stored in one memory, while the data to be processed is stored in another. This allows the processor to fetch a value from program memory and one from data memory at the same time, in a single memory cycle. Consider the effect of this on the FIR filter algorithm. To implement each tap of the filter, the program must multiply a constant value (the filter coefficient) by a data vale (the stored sample value). The processor can fetch both values from memory simultaneously, saving one memory cycle. When large filters are being implemented, the savings can quickly mount. And typically, the processor can perform the needed multiplication, subsequent addition of the product to an accumulator, and shifting of the data value in the storage array in a single machine cycle. Contrast this with the many cycles needed to perform the same operations in a general-purpose computer and you can see why specialized processors are so much more capable of processing sampled signals. DSP chips also often include other optimizations, such as *pipelining* of instructions and specialized addressing modes to support FFT operations.

Fixed Point vs Floating Point

One of the things that makes general-purpose computers so useful is their ability to perform *floating-point* calculations. Floating-point representation of numbers treats the stored value as a fraction (the *mantissa*) of magnitude less than 1 and an exponent (usually base 2). This approach allows the computer to handle a great range of numbers, from the very small to the very large. Some modern DSP chips support floating-point calculations, too. But this is not as great an advantage for signal processing as it is for general-purpose computing because the range of values needed in DSP is fairly small. For this reason, *fixed-point* processors are common for DSP.

A fixed-point processor treats a stored value as just the mantissa part—there is no exponent. This does not mean that only fractional numbers can be handled. The radix point—the separation between the integer and fractional parts of a number—can be between any two bits of the stored number. Indeed, the selection of a radix point is somewhat arbitrary. But having a fixed radix point does complicate things somewhat for the programmer. When multiplying two numbers, let's say they are 16-bit values, the resulting number has twice as many bits—32, in this case. And where the radix point falls in those 32 bits depends on where it was in the original numbers. If the 16 bits were composed of three bits of integer value, followed by 13 bits of fractional value, the 32-bit product would have 6 bits of integer value and 26 bits of fraction. That means that to store the upper part of the product as a 16-bit value, the product has to be shifted left three bits. Because of this, fixed-point DSP chips often include special shift hardware that allows shifting of the data during load and store instructions. The programmer must ensure that the proper shift values are part of the instruction. It is also imperative that the product not overflow the three least-significant bits of integer value. Keeping all of this straight becomes a headache when programming a fixed-point processor. Still, because fixed-point processors are simpler—and thus less

expensive—they are common in low-cost DSP systems. **Table 18.2** shows some common DSP chip families.

Table 18.2
Some Common DSP Chips

Manufacturer	*DSP chip*	*Word size (bits)*
Fixed-point		
Analog Devices	ADSP-2100 family	16
Motorola	DSP56000 family	32
Texas Instruments	TMS320 family	16
Floating Point		
Analog Devices	ADSP-21020	32
Motorola	DSP96002	32
Texas Instruments	TMS320 family	32

DEVELOPMENT TOOLS

Developing DSP systems for Amateur Radio requires the right development tools. Included in these are the hardware that includes a DSP chip, A/D and D/A converters, input and output low-pass filters, and some means of communicating with a PC for loading and testing programs. An assembler and/or a high-level language compiler are needed as well. Debugging software is desirable, too. While industrial-grade DSP development platforms abound, their cost is prohibitive for the amateur. Low-cost development tools are needed. Recently, such tools have become available.

The Texas Instruments DSP Starter Kits

Texas Instruments provides a DSP Starter Kit (DSK) for both its TMS320C25-series and TMS320C50 series 16-bit, fixed-point processors. Each kit consists of a small PC board that contains the processor, with embedded ROM bootstrap firmware, an audio-frequency codec (integrated A/D, D/A and low-pass filters), and the needed power-supply circuitry, except for the transformer. Also included are a simple assembler and a debugger. Kits are sold by all TI distributors. Call 800-336-5236 for one near you.

The DSK, which costs about $100, does not use low-grade processors; these are prime, state-of-the art processors, capable of executing just about any audio-frequency algorithm amateurs are likely to want. The kits also include connector holes (but not the connectors) for attaching external peripheral devices to the processor bus. Although labeled a kit, this product requires no assembly.

THE TAPR DSP-93

Tucson Amateur Packet Radio, Inc, a not-for-profit organization devoted to the advancement of packet radio, has the DSP-93 kit available. This kit, which requires assembly, is based on the TMS320C25 processor and consists of stackable boards that include 32 kwords of program and data memory, with space for 64 kwords. The analog board includes a codec capable of sampling at 45 ksamples/s, an analog multiplexer to select from up to 8 audio input sources and an interface for radio keying and frequency-control lines, as well as an RS-232 interface for computer communication. The stackable feature permits later addition of other accessory boards, for use with different analog subsystems, faster interfaces to a PC, or specialized analog circuitry. A shareware assembler is available to facilitate development.

PC SOUND CARDS

Perhaps one of the most enticing ways of putting DSP development tools in the hands of amateurs is the PC sound card. Recent additions to the market include sound cards with embedded DSP chips. For amateurs, finding the development tools for such boards is the challenge. Sound card manufacturers typically make development packages available, but not at low cost. One set of free tools, and a description of the Analog Devices Personal Sound Architecture chip set used on some of these boards, is described in "Programming a DSP Sound Card for Amateur Radio," by Johan Forrer, KC7WW, *QEX*, August 1994.

BIBLIOGRAPHY

(key: **D** = Disk included, **A** = Disk available, **F** = Filter design software)

Software Tools Books

O. Alkin, *PC-DSP*, Prentice Hall, Englewood Cliffs, NJ, 1990. (**DF**)

A. Kamas and E. Lee, *Digital Signal Processing Experiments*, Prentice Hall, 1989. (**DF**)

S. D. Stearns and R. A. David, *Signal Processing Algorithms in FORTRAN and C*, Prentice Hall, 1993. (**DF**)

DSP Textbooks

D. DeFatta et al, *Digital Signal Processing: A System Design Approach*, Wiley, New York, 1988.

M. E. Frerking, *Digital Signal Processing in Communication Systems*, Van Nostrand Reinhold, New York, 1994.

E. Ifeachor and B. Jervis, *Digital Signal Processing: A Practical Approach*, Addison-Wesley, 1993. (**AF**)

A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing*, Prentice Hall, 1975.

T. Parsons, *Voice and Speech Processing*, McGraw-Hill, Hightstown, NJ, 1987.

J. Proakis and D. Manolakis, *Digital Signal Processing*, Macmillan, New York, 1988.

L. R. Rabiner and R. W. Schafer, *Digital Processing of Speech Signals*, Prentice Hall, 1978.

Articles

J. Albert, "A New DSP for Packet," *QEX*, Jan 1992, pp 3-5.

J. Albert and W. Torgrim, "Developing Software for DSP," *QEX*, Oct 1992, pp 3-6.

P. T. Anderson, "A Simple SSB Receiver Using a Digital Down Converter," *QEX*, Mar 1994, pp 17-23.

J. Ash, et al, "DSP Voice Frequency Compandor for use in RF Communications," *QEX*, Jul 1994, pp 5-10.

B. Bergeron, "Digital Signal Processing Part 1: The Fundamentals," *Ham Radio*, Apr 1990, pp 24-35; Part 2, *Communications Quarterly*, Fall 1990, pp 45-54; Part 3, *Communications Quarterly*, Spring 1991, pp 23-36; Part 4, *Communications Quarterly*, Winter 1991, pp 77-87.

J. Bloom, "Measuring SINAD Using DSP," *QEX*, Jun 1993, pp 9-18.

J. Bloom, "Negative Frequencies and Complex Signals," *QEX*, Sep 1994.

H. Cahn,"Direct Digital Synthesis—An Intuitive Introduction," *QST*, Aug 1994, pp 30-32.

B. de Carle, "A Receiver Spectral Display Using DSP," *QST*, Jan 1992, pp 23-29.

B. de Carle, "A DSP Version of Coherent CW," *QEX*, Feb 1994, pp 25-30.

A. Dell'Immagine, "Digital Filter for EME Applications," *QEX*, May 1992, pp 3-6.

D. Emerson, "Digital Processing of Weak Signals Buried in Noise," *QEX*, Jan 1994, pp 17-25.

J. Forrer, "Programming a DSP Sound Card for Amateur Radio," *QEX*, Aug 1994.

B. Hale, "An Introduction to Digital Signal Processing," *QST*, Jul 1991, pp 35-37.

D. Hershberger, "Low-Cost Digital Signal Processing for the Radio Amateur," *QST*, Sep 1992, pp 43-51.

D. Hershberger, "A Digital Signal Processor" in the **Filters** chapter.

F. Morrison, "The Magic of Digital Filters," *QEX*, Feb 1993, pp 3-8.

C. Puig, "A Weaver Method SSB Modulator Using DSP," *QEX*, Sep 1993, pp 8-13.

R. Olsen, "Digital Signal Processing for the Experimenter," *QST*, Nov 1984, pp 22-27.

H. Price, "Digital Communications: Audio Spectrum Analyzers—Cheap," *QEX*, Dec 1993, pp 13-15.

S. Reyer and D. Hershberger, "Using the LMS Algorithm for QRM and QRN Reduction," *QEX*, Sep 1992, pp 3-8.

R. Ward, "Basic Digital Filters," *QEX*, Aug 1993, pp 7-8.