Ray Mack, W5IFS

17060 Conway Springs Ct, Austin, TX 78717; **w5ifs@arrl.net**

SDR: Simplified

More Filter Activites

FIR Filter Recap

Figure 1 shows a graphical representation of a 5 tap FIR filter. The coefficients are those calculated for a low pass filter with a cut-off frequency of 1100 Hz with 8000 samples/s and 40 dB stop band attenuation. I used the FIR_filter_generator.exe program from the last installment.[1] Table 1 shows the coefficients in 16 bit integer and floating point representation. The formula for the filter is expressed as:

$H(z) = 0.02316\ z^0 + 0.16647\ z^{-1} + 0.27499\ z^{-2} + 0.16647\ z^{-3} + 0.02316\ z^{-4}$

I have always found "z" notation to be very confusing. It is a lot easier to visualize with a picture such as Figure 1, where each "z" value is just the sample contained in one of the shift register positions. H(z) is just what comes out of the adder at the bottom of the filter in Figure 1. H(z) is the sequence of numbers that are calculated by our DSP. We put H(z) into a DAC and low pass filter, and get H(t) which is now a continuous time function that we can hear through a speaker or watch on an oscilloscope. The notation "H(z)" is the form you will commonly see in an engineering text for the response of a filter.

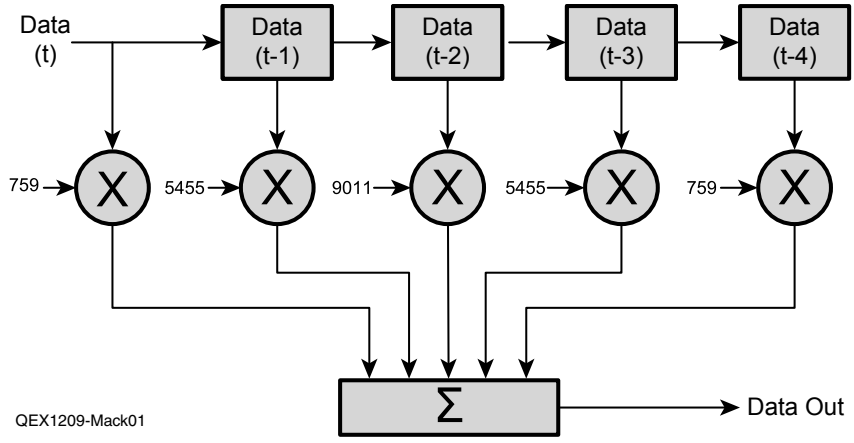


Figure 1 — This diagram is a z-space representation of a simple 5 tap FIR filter.

Filter Response Calculation

That is all very interesting, but it is not terribly useful for figuring out if our filter is really going to do what we want. The Fourier series that corresponds to our filter is what describes the actual frequency response for all possible input frequencies. This is similar to what happens when we use the Fourier series to create a square wave:

$G(t) = \sin(2\pi ft) + 1/3 \sin(3 \times 2\pi ft) + 1/5 \sin(5 \times 2\pi ft) + 1/7 \sin(7 \times 2\pi ft) + \ldots$

If you add up all of the harmonics, you get an exact square wave. If you stop after harmonic 19, for instance, you get a waveform that is close to a square wave but shows the Gibbs phenomenon. (You might want to use *Octave* or *Gnuplot* to see what happens.) This process takes information in the frequency domain (1, 1/3, 1/5, …) and converts it to a time domain representation. We have done an inverse Fourier transform to transform frequency domain information to time domain information.

In order to get a true representation of the filter frequency response, we just need to replace each "z" with a representation of the Fourier series for the filter. The task we are doing is a Fourier transform. Remember that the filter coefficients describe the operation of the filter in the time domain and a Fourier transform changes time domain information into frequency domain information. The transform of our filter looks like this:

$H(f) = 0.02316 (\cos(0 \times 2\pi f) - j \sin(0 \times 2\pi f)) + 0.16647 (\cos(1 \times 2\pi f) - j \sin(1 \times 2\pi f)) + 0.27499 (\cos(2 \times 2\pi f) - j \sin(2 \times 2\pi f)) + 0.16647 (\cos(3 \times 2\pi f) - j \sin(3 \times 2\pi f)) + 0.02316 (\cos(4 \times 2\pi f) - j \sin(4 \times 2\pi f))$

Since cos (0) is one and sin (0) is zero, this simplifies to:

$H(f) = 0.02316 + 0.16647 (\cos(1 \times 2\pi f) - j \sin(1 \times 2\pi f)) + 0.27499 (\cos(2 \times 2\pi f) - j \sin(2 \times 2\pi f)) + 0.16647 (\cos(3 \times 2\pi f) - j \sin(3 \times 2\pi f)) + 0.02316 (\cos(4 \times 2\pi f) - j \sin(4 \times 2\pi f))$

Remember that each $\cos(x \times 2\pi f) - j \sin(x \times 2\pi f)$ is just a single sine wave represented in rectangular form (x and y) rather than polar form (amplitude and phase angle). In signal processing, we actually refer to rectangular form as I (the cos term) and Q (the sin term) rather than what we did in algebra class with x and y.

Putting the Software in SDR

I did an on-line search and found no program that automates the process of calculating a set of filter coefficients and displaying the resulting filter response. That doesn't mean one doesn't exist, but it does not show up in a search. *MATLAB* and *Octave* each have a function that will compute the frequency response if you give it an array holding the filter values. Both require a fair amount of programming to do the computations and display. I find both as incomprehensible as most DSP math!

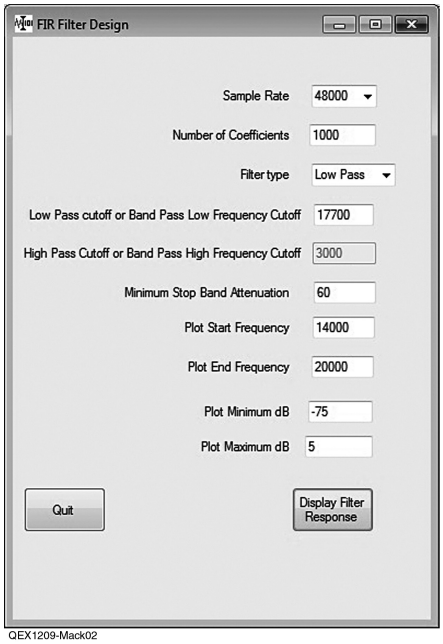
Table 1
Coefficients for a 5 TAP FIR Filter

Coefficient	*16 bit Integer*	*Floating Point*
0	759	0.02316
1	5455	0.16647
2	9011	0.27499
3	5455	0.16647
4	759	0.02316

Since no program exists, I have created a program that incorporates the filter calculations with a Kaiser window and then displays the response. Figure 2 shows the main window for the program with representative values filled in. It allows you to enter the same information as the *C* program from the May/June column. Figure 3 shows a representative output window for the program. The program is available on the ARRL *QEX* files website, and the source code is included.[2]

I wrote the program in *Visual Basic 2010 Express* because it is the easiest environment I know to write a *Windows* program. This is not your father's *BASIC*. It isn't even very much like *Visual Basic 4* (the last one I used regularly) or *Visual Basic 6*. In 2008, Microsoft did a major re-

[1]Notes appear on page 37.



QEX1209-Mack02

Figure 2 — This screen shot is the main window of the filter response program.

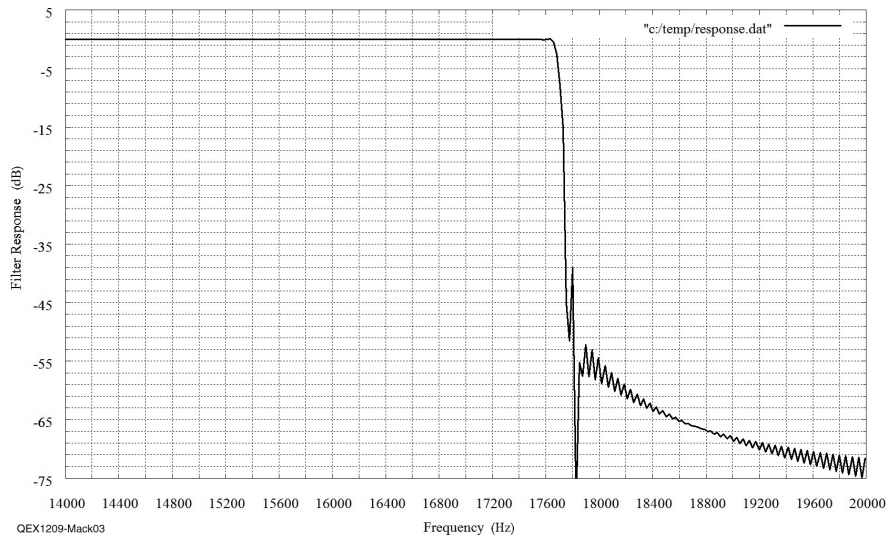


Figure 3 — Here is a representative output window of the filter response program.

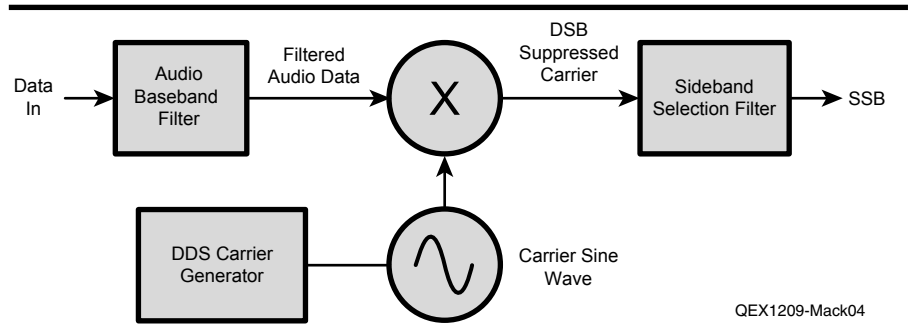


Figure 4 — This block diagram shows the software SSB transmit generator using the filter method.

write of *Visual Basic* to make it significantly more object oriented and to incorporate the .NET framework as an integral part of the language. Perhaps the biggest change from all previous *BASIC* systems is that all arrays start at index zero instead of index one. If you are a *C* programmer, you will find the new *Visual Basic* to be a minor change in syntax. It took me about 10 minutes to modify the syntax of the *C* console program from the last column to do the Kaiser window and coefficient calculations in *Visual Basic*.

The .NET features for doing *Windows* programs are a real boon for writing programs with one exception. I find the Chart tool to be totally incomprehensible. I was able to get a barely useful X-Y plot of the filter response after 3 days of fighting the many layers of parameters. I figured out how to do the same tasks in *Gnuplot* in about 3 hours when I first started learning that tool. Fortunately, *Gnuplot* comes with an executable image that can be run from another program to simply pop up a new window. I have incorporated that mechanism into our program. The program can still use some improvement to make it easier to have *Gnuplot* do the display. It is left as an "exercise for the reader" until I get a chance to get back to improve it. For now, we need to get back to making a radio!

An SSB Transmit Generator

Now that we have the ability to create a sine wave using the DDS program and to create a sharp cutoff filter, we can create a filter method SSB generator. The structure of our program is the same as if it were implemented in analog hardware. Figure 4 shows the block diagram of the system. The program operates in a serial fashion: first the baseband filter limits the audio to a band of 300 Hz to 3 kHz, second it computes the phase value for the carrier, third is the multiplication for the balanced mixer, and finally the undesired sideband is removed. The filter response program shows that the low audio cutoff is only useful with 200 taps or more. At 100 taps the rejection is only on the order of 12 dB below 100 Hz. Likewise, the opposite sideband filter requires on the order of 700 to 1000 taps to give approximately 60 dB of opposite sideband suppression. The large number of taps also makes the skirts very steep, so that we can use the filter to also further reduce any carrier feed through.

There are a number of compromises we could make if we were going to make a real transmitter. The first is setting the lower frequency limit for audio. Simply using a dc block in the analog portion of the audio chain will set a lower boundary on the frequency. The response will be zero at 0 Hz and rise very rapidly to the frequency we set. This reduces the need for a sharp cutoff in DSP. The close in rejection of audio above 3 kHz is 45 dB or more. Additionally, there is almost no energy above 3 kHz in the human voice, so energy in that region will likely be at least 60 dB below the lower frequencies after filtering. Limiting the higher frequencies allows us to use a 6 kHz wide sideband selection filter instead of the normal 3 kHz filter to get better skirt response. A low pass or high pass filter would also work and give approximately the same skirt response, but we want to be sure to eliminate any residual energy at baseband in the case of a lower sideband transmission. The wider bandwidth limits the lower frequency for our carrier. We want the carrier frequency to be as high as possible in order to limit image response when we up convert to our final RF signal. This experimental transmitter is not really suitable as a real transmitter because the CODEC limits the highest frequency to 20 kHz. It is truly just an audio CODEC. When I have more time, I would like to go back to the Blackfin Stamp so that I can use the DAC08 at 1 MHz sample frequency for transmit and build an ADC board that can also sample near 1 MHz. My goal is to make a 6 m sideband rig to fill in

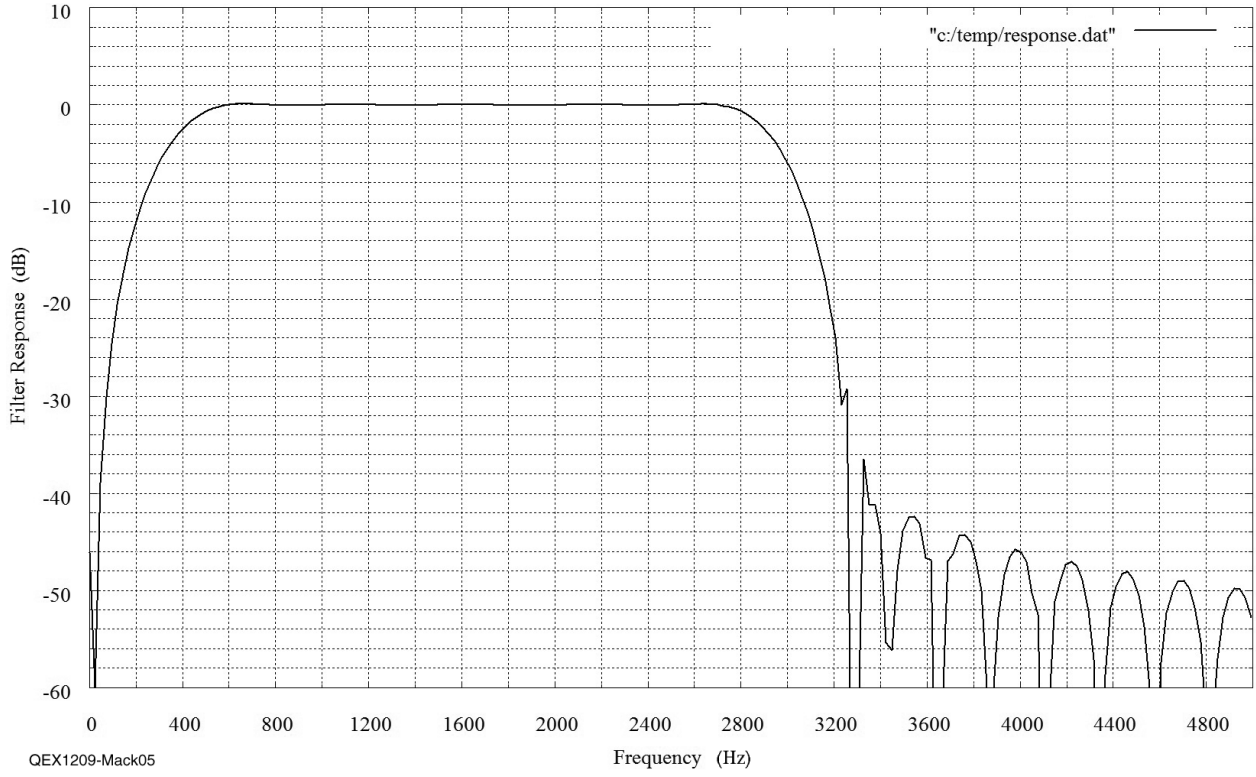


Figure 5 — This graph is the response of the 200 tap baseband filter.

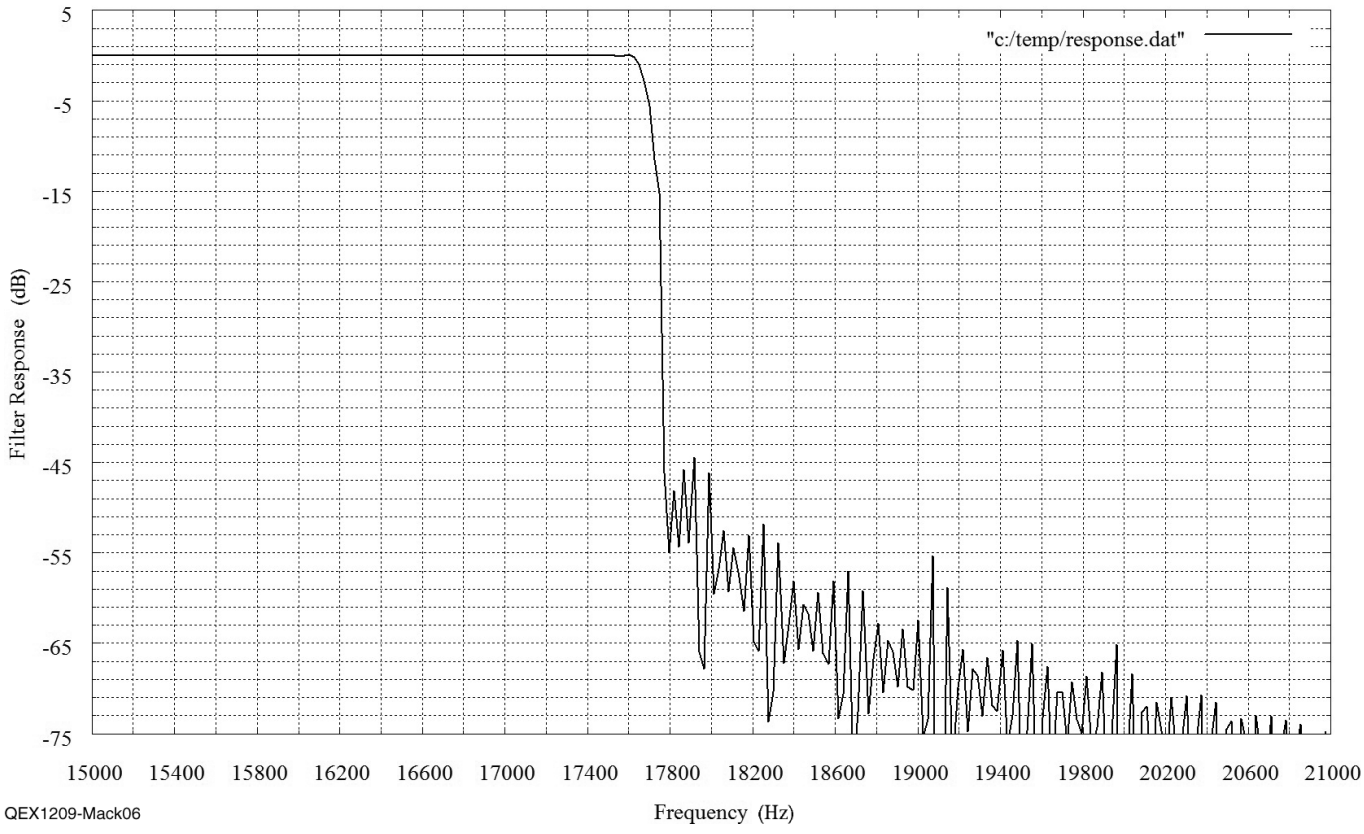


Figure 6 — This graph is the response of the 700 tap sideband selection filter for a carrier at 18 kHz. It shows the response of the opposite sideband.

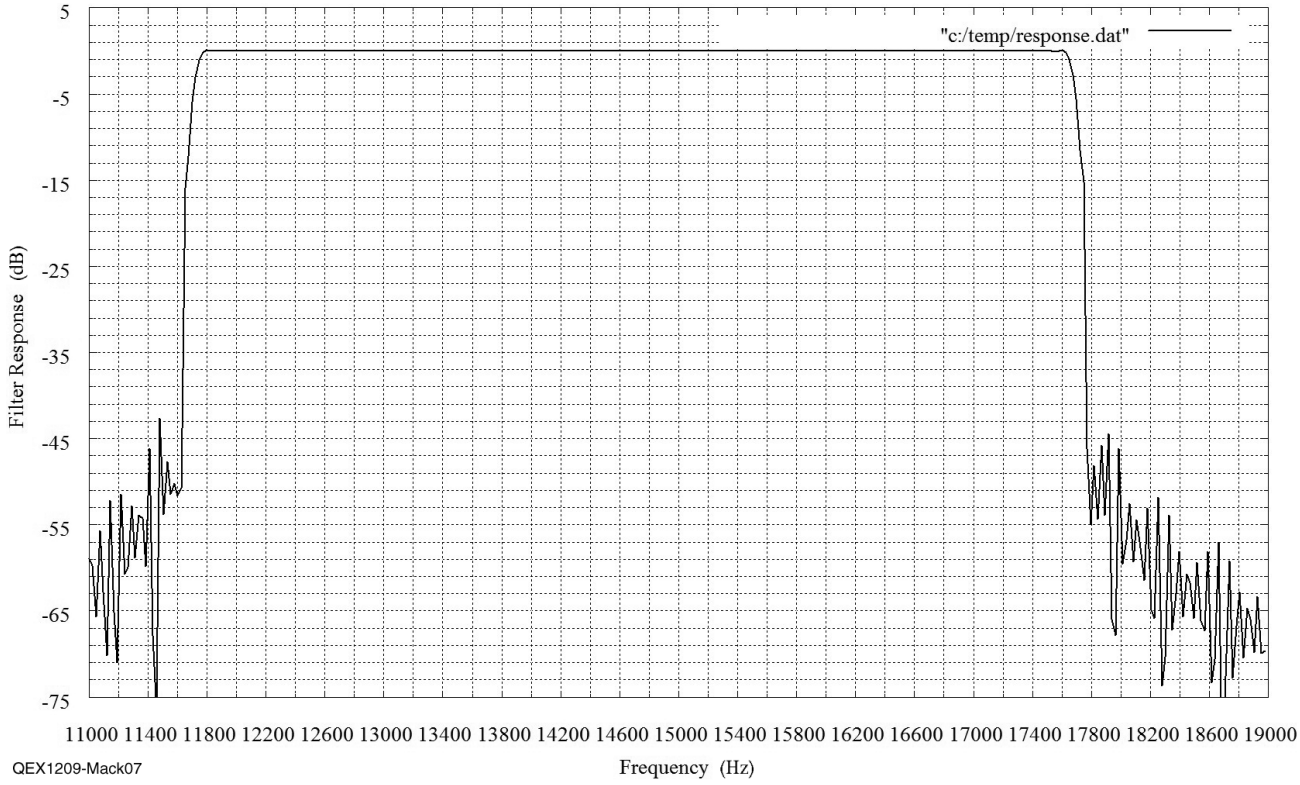


Figure 7 — Here is the response of the 700 tap filter, showing a wider frequency view. The filter is 6 kHz wide to allow for a steep skirt on the carrier side. The 6 dB cutoff point is set to 300 Hz away from the carrier.

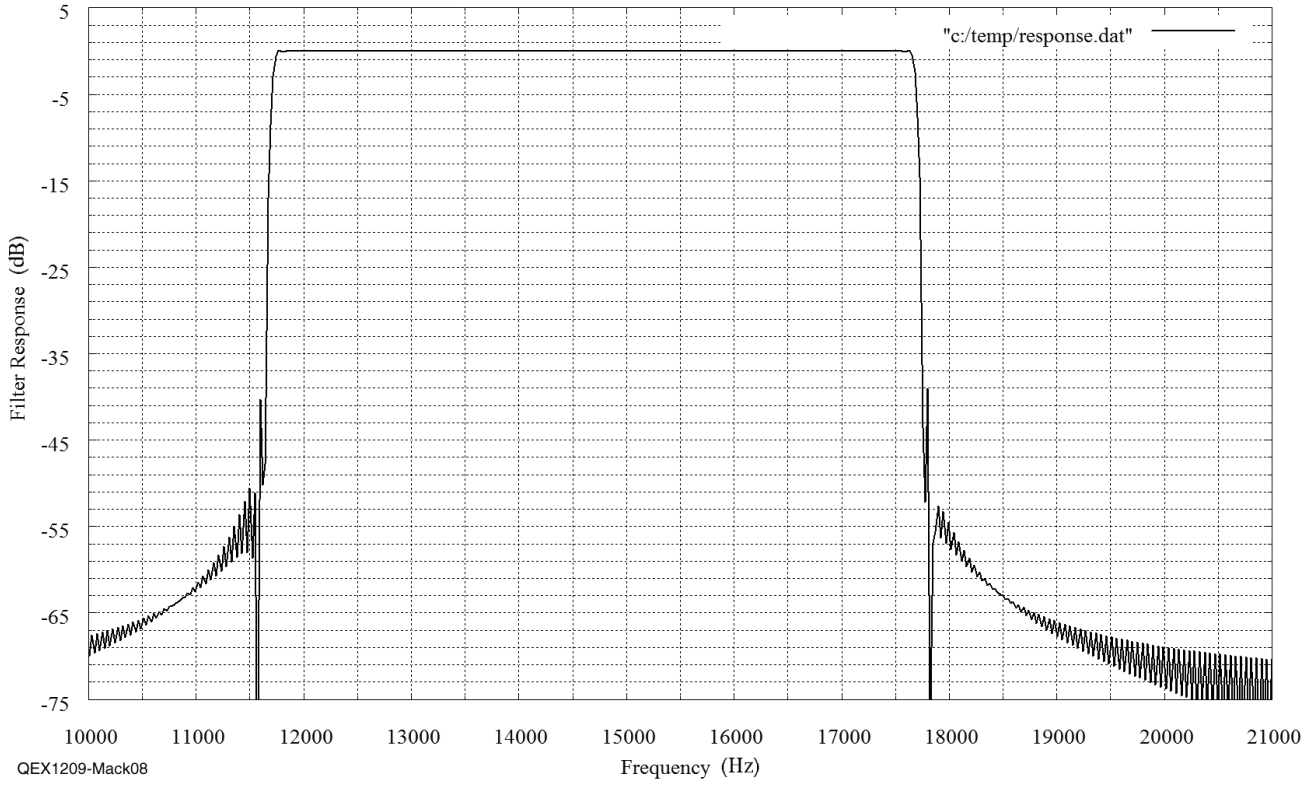


Figure 8 — The response of a 1000 tap filter is represented in this graph. It shows the trade off of more calculations versus out of band rejection.

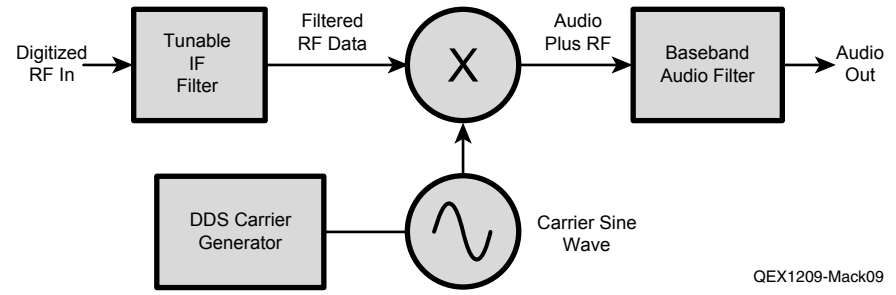


Figure 9 — Here is a block diagram of the SSB receiver using DSP.

one of the holes in my station.

We have a practical limit with respect to the number of taps in the filters. Each tap requires one multiply-accumulate operation, which is an MAC in the DSP world. (Unfortunately, MAC also means Media Access Control if you are a networking hardware person and an object lesson in why three letter initials can be a bad idea!) The DSP is capable of one MAC for each MHz of clock frequency for each portion of the hardware chain. The C5535 has two multipliers in the chain, so it is capable 200 million MACs (TI uses the initials MMAC) if the processor is running at 100 MHz and the library software uses both multipliers. Our experiments are running the CODEC at a 48 kHz sample rate, so our transmitter with 200 taps for audio and 1000 taps for sideband selection will need 57.6 MMACs to do its job. Regular software also requires one or two instructions per clock cycle. If we take the worst case of one multiplier used and one instruction per cycle, we have approximately 40 per cent of the DSP left over for regular computing with a 100 MHz clock. The best case is 70% of capacity left over.

An SSB Receiver

Figure 9 shows how we can reverse the steps above to create an SSB receiver. We simply run the "RF" from a down converter into one channel of the CODEC. We set the "RF" filter frequency to select the desired signal, multiply it with the output of our DDS generator, and then filter the resulting audio. The audio filter produces better audio if it is a band pass filter than just a low pass filter because the RF filter will allow some opposite sideband energy to pass at very low frequencies.

Odd and Even Functions

We need to get a little closer to the math in preparation for dealing with the 90° phase shift that is important to a lot of DSP operations. At the beginning of this column, we looked at the Fourier series for our filters in both the time domain and the frequency domain. The series is a general case where the phase and amplitude are arbitrary. The case of a square wave is more than just a curiosity. Figure 10 shows two different square waves with appropriate time scale. They are identical in frequency and amplitude and both extend from negative infinity to positive infinity. They differ in phase by 90°, though. The top waveform is called an even function because the value of the waveform at 1 second is the same as the value at –1 second. The bottom waveform is called an odd function because the value of the waveform at –1 second is equal to the value at 1 second but multiplied by –1. This has implications for the Fourier series for the two waveforms. The even function has the Fourier series:

$G(t) = \cos(2\pi t) - \frac{1}{3}\cos(3 \times 2\pi t) + \frac{1}{5}\cos(5 \times 2\pi t) - \frac{1}{7}\cos(7 \times 2\pi t) + \ldots$

The odd function has the Fourier series:

$G(t) = \sin(2\pi t) + \frac{1}{3}\cos(3 \times 2\pi t) + \frac{1}{5}\cos(5 \times 2\pi t) + \frac{1}{7}\cos(7 \times 2\pi t) - \ldots$

Our implementations of FIR filters have always been even functions in the frequency domain. For that reason, the filter coefficients have always been mirror images around the center. In the 5 tap example, tap one was the same as tap three and tap zero was the same as tap four. Since it is an odd size, you can think of tap 2 being the same for zero and "minus zero" since it is exactly in the center. The coefficients have exact mirror image pairs for an even number of coefficients. Figure 11 shows the ideal filter response of the 5 tap filter for positive and negative frequencies.

Mathematicians call certain phenomena "degenerate cases." A point is a degenerate case of a circle: it has a radius of 0. In DSP we have a degenerate case called

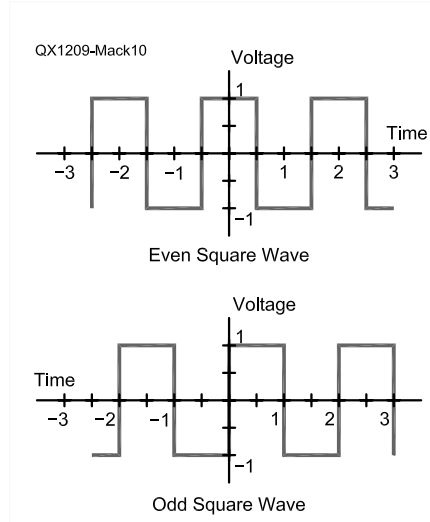


Figure 10 — Odd and even square waves in the time domain.

an all pass filter. It passes all frequencies with unchanged amplitude. There are two cases of all pass filters, however: one is odd and one is even! Figure 12 shows the two responses.

We generate the filter constants for an FIR by taking the Fourier transform of the frequency response. The transform of our even all pass filter is another degenerate case. To get out exactly what you put in, you just multiply each sample by one (cos 0). This is shown in the top of Figure 13, where we end up with a single coefficient. When we put in a cosine wave we get the cosine wave back out. Our odd all pass filter has a Fourier series that contains only sine terms rather than cosine terms. Those coefficients correspond exactly to our Fourier series for the odd square wave. The positive values are 0, ⅓, 0, ⅕, 0, ⅐ … Since it is an odd function, the values for our DSP implementation will be –⅐, –0, –⅕, –0, –1/3, –0, –1, 0, 1, 0, ⅓, 0, ⅕, 0, ⅐.

The important implication for the odd function all pass filter is that putting in a cosine wave at any frequency will produce a sine wave with the exact frequency of the

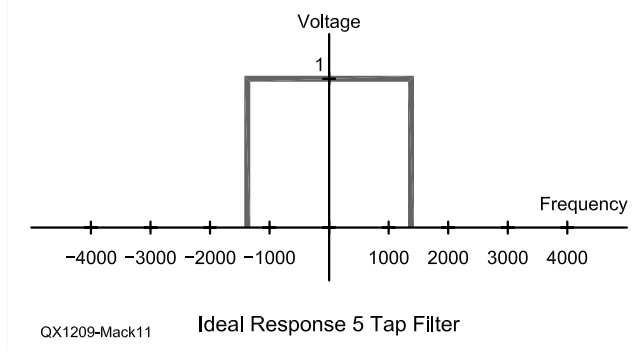


Figure 11 — This is the Ideal filter response for the 5 tap filter, showing its even order characteristic.

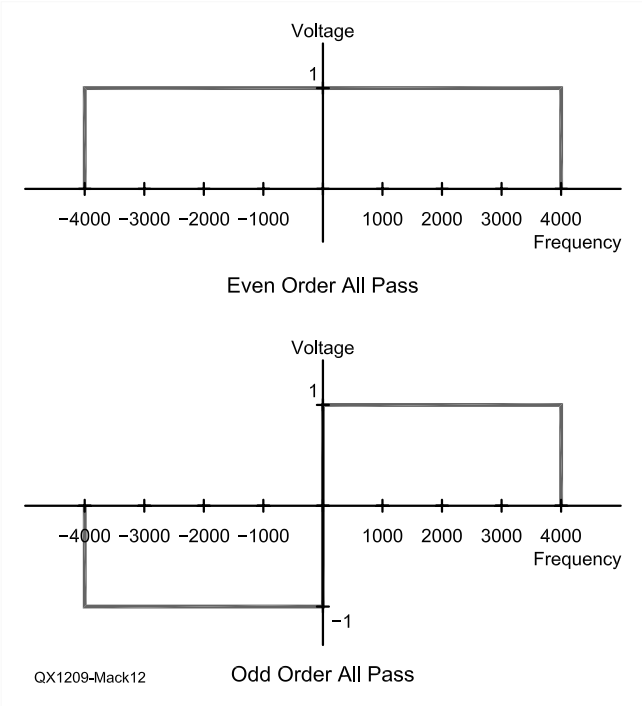


Figure 12 — Here are even order and odd order all pass filters.

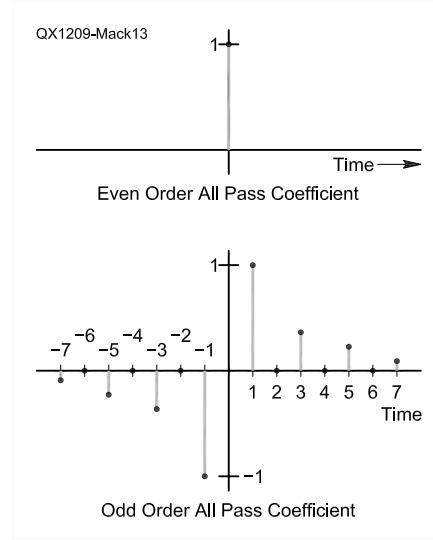


Figure 13 — The filter coefficients for even order and odd order all pass filters are shown on this graph.

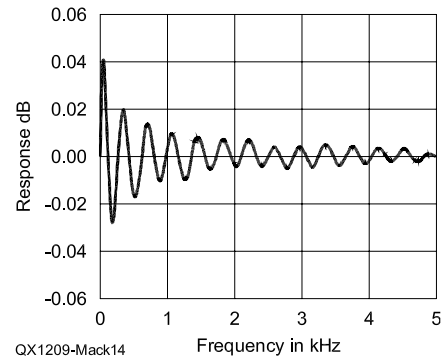


Figure 14 — This is the amplitude response of a 247 tap Hilbert Transform with a 48 kHz sample rate.

input. This is an exact 90° phase shift! The odd order all pass filter is called a Hilbert Transform when implemented in DSP.

The exact 90° phase shift is exactly what we need for many RF signal generation tasks. There is no free lunch, however! Remember that there is that nasty discontinuity in the frequency response at dc. We saw before that any sharp change in the frequency domain causes the Gibbs phenomenon to appear. It is the same here. We get a constant phase shift, but the amplitude is not constant and rings at the frequency of the discontinuity (0 Hz and ½ fs in this case). We saw before that increasing the number of filter coefficients to a very large number will compress the ringing in the frequency response to a small portion of the total, but will not eliminate the 8% overshoot. A useful Hilbert transform will require a large number of coefficients to move the bulk of the amplitude error below our lowest frequency of interest. Another limitation of the Hilbert transform is that it requires an odd number of coefficients.

The Phasing Method

Chapter 11of *Experimental Methods in RF Design* presents a good description of the use of the phasing method in DSP for an 18 MHz transceiver.[3] Chapter 9 describes the theory of the phasing method with equations to show how amplitude error and phase error affect opposite sideband suppression.

The heart of the phasing method is the same amplitude but 90° phase difference between the two channels. Any deviation from exactly 90° and any amplitude imbalance cause less than perfect opposite sideband suppression. The rule of thumb is that 0.1 dB of amplitude imbalance plus one degree of phase error will limit sideband suppression to –40 dB. Those limits are representative of what is possible with analog components with temperature fluctuations and normal component variations. DSP eliminates the issues with component changes from ideal. We get response that is only limited by the precision of the ADC and DAC and the number of taps we choose to implement. Since a Hilbert Transform does not have phase error, the opposite sideband suppression is determined solely by amplitude imbalance. Figure 14 recreates the example from Chapter 11 and shows that a 247 tap filter at 48 kHz sample rate will have 0.02 dB amplitude imbalance near 300 Hz. That will yield opposite sideband suppression of 52 dB.

Every other coefficient of the Hilbert Transform is zero, as is the center, so a 247 tap transform will only need 123 multiply-accumulate operations if implemented efficiently. This is a significant savings over the filter method example given earlier, which required 1000 multiply-accumulate operations for equivalent performance. The TI library contains a Hilbert transform function, but it is not clear if it implements a cycle saving algorithm different from an FIR filter. The software for this issue does not contain any phasing examples. I hope to include that next time.

Notes

[1]Ray Mack, W5IFS, "SDR:Simplified, Filter Design Program," May/Jun 2012 *QEX*, pp 40-44. The software files described in that column are available for download from the ARRL *QEX* files website. Go to **www.arrl.org/qexfiles** and look for the file **5x12_Mack_SDR.zip**.

[2]The software for this column is available for download from the ARRL *QEX* files website. Go to **www.arrl.org/qexfiles** and look for the file **9x12_Mack_SDR.zip**.

[3]Wes Hayward, W7ZOI, Rick Campbell, KK7B, and Bob Larkin, W7PUA, *Experimental Methods in RF Design*, The American Radio Relay League, 2009, ISBN: 978-087259-923-9. ARRL Publication Order No. 9239, $49.95. ARRL publications are available from your local ARRL dealer or from the ARRL Bookstore. Telephone toll free in the US: 888-277-5289, or call 860-594-0355, fax 860-594-0303; **www.arrl.org/shop**; **pubsales@arrl.org**.

QEX