Making a Software Defined Radio for the QRP Enthusiast—Part III

Ward Harriman—AE6TY ae6ty@arrl.net

In the last issue of this magazine, I began to describe the software used in my self-contained (no PC) Software Defined Radio (SDR). Major points of this description were the integrated development environment, board support, user interface, and generation of a single sideband signal. At that point, the receiver could be tuned and a sideband selected. However, the receiver still lacked the filters necessary to a practical transceiver. A discussion of the methods used to implement those filters and the things I learned along the way toward that implementation will form this part of my article.

There are many different ways to do filtering and even a cursory discussion of the major players is beyond the scope of this paper. For the purposes of this article I will describe two ways I've implemented filtering. But before I describe how I did my filtering, I'm going to need to describe two of the most fundamental principles of Digital Signal Processing: Correlation and Fourier Transforms. No… don't stop reading… I'm going to try to present each of these concepts in a very informal matter. I'll gloss over all of the math and all the nitty-gritty and try to show you this stuff can be intuitive and very powerful………

In reading through the literature I often heard the terms "in the time domain" and "in the frequency domain." I have learned that at best these terms are hints as to the view point of the author and convention. In theory there is no difference between "time domain" and "frequency domain" and that means you are free to think about the problem in either way. Indeed, the only reason one thinks "frequency" instead of "time" is one of convenience and simplification. For example, if you are like me, you tend to think of filters as working in the "frequency domain." I think, "I want to pass some frequencies and block others." Thus, when thinking about designing filters I want to think in "frequency." However, CODEC samples come periodically in "time," and so when implementing filters I may want to program in "time."

This ability to work "in the frequency domain" is hugely powerful. Now I'm going to exaggerate a little… I can say "remove any 1 kHz component" and get exactly that. I can say "boost all frequencies below 1500 Hz by 20%" and get exactly that. I can say, "change the phase of the 750 Hz component of the signal." All these capabilities come from a single technology: the Discrete Fourier Transform. But to get to the DFT you really need to start with something I found even less familiar: *correlation*.

Now over the last few years I've tried to understand digital filters and how to explain them to others without using too much math. What follows are my best attempts to date. I am going to brush over some pretty important "secondary" effects. I do this consciously so as to deliver the material in an intuitive and natural progression. As you read further on this topic you will no doubt find critical details I have left out. When you do, please accept my apologies. Let's get started.

Correlation

What is "correlation"? Simply put, correlation is a method used to measure similarities—"How alike are A and B?" Suppose A and B are collections (arrays) of CODEC samples each 100 samples long. Suppose further that B is an array which was created by sampling a 600 Hz signal. If I then compare A and B and find them "very similar," then I know there is a lot of 600 Hz signal in A. If they are not similar at all, then I know there isn't any 600 Hz in A. The result of a correlation is a numeric value we'll call H. We'll say H=1 if the A array looks exactly like the B array and H=–1 if A is the negative of B. We'll say H = 0 when A is nothing like B. H can thus take on any value –1<H<1.

(NOTE: When writing about these issues it is convenient to develop a terminology to simplify writing and reading. For example, we will need to refer to arrays of numbers. When I want to refer to the "B Array" I will write "B[ ]"… the "[ ]" can be read directly as "Array." When I want to talk about an array which contains a signal with a specific frequency I'll write "B(freq)[ ]." As an example, "B(600)[ ]" is the "B Array containing 600 Hz." (Note that B(600)[ ] is not really a recording of a 600 Hz signal; the contents of B(600)[ ] are computed using the sine or cosine functions provided by Microchip.)

Correlation can be used to implement a filter. To do so, I compare A to B to generate H. Correlation is very fast and I can compute a new H every CODEC sample period. I can then send H directly to the CODEC for output. Yes, directly. To see why, consider the following. When the signals in A and B are both 600 Hz and "in phase" then H will be a 1; A is equal to B. During the next CODEC sample the phase of A[ ] will be different and A will look "mostly like" B and so H will be somewhat less than 1. As time progresses A will eventually look like the negative of B and H will be –1. Again, time progresses and A will again look exactly like B and H will be 1 again. In fact, H will cycle from 1 to –1 and back to 1 at 600 Hz. Yep, a filter. A more in depth presentation of this material can be found at http://www.ae6ty.com/Intro2SDR/SDRnoMath.html.

Now just exactly how does correlation compare A to B? Well, it is really quite simple. Correlation multiplies each element in A by the corresponding element in B and then adds up the products. Stated a little more succinctly:

H = SUM(A[i] x B[i]) for all i

We can write the program for correlation in C. Loops in C are written using a "for" statement. For A[ ] and B[ ] arrays which are 100 elements long, we would write:

```
H = 0;                        //initialize our sum.
for (i=0;i<100;i=i+1)         //for all i in A[ ] and B[ ]
H = H + A[i]*B[i];            //multiply and accumulate.
```

Now before we proceed I want to drive home how important this is. Correlation allows me to compare two signals and get a numerical figure of merit. This simple concept is the basis for everything that follows.

"But," you say, "Suppose I want a filter which passes more

than a single frequency? Suppose I'd like to pass frequencies 600 and 650?" Well, then we would do a correlation with B(600Hz) and a second for B(650Hz) and then add the two together. Thinking ahead you can see a problem: for a wide filter I'd need to do a whole bunch of correlations and that sounds like a lot of processing time and it WOULD BE but, of course, there is a short cut.

Instead of performing a bunch of correlations and then adding the results I could add the results along the way. I would write the code:

```
H = 0;
for (i=0;i<256;i=i+1)
  {
      H = H + A[i]*B(600Hz)[i];        // 600 Hz.
      H = H + A[i]*B(650Hz)[i];        // 650 Hz.
  }
```

This reduces the loop overhead but it is still an awful lot of multiplies. Fortunately, there is yet another simplification. I can add all the B(600Hz)[ ] and B(650Hz)[ ] arrays together. After all, high school algebra says that ab+ac=a(b+c) and so H = A*B + A*C = A(B+C). We can call this new array F[ ] and make F[ ] = B(600Hz)[ ] + B(650Hz)[ ]. Then instead of doing 2 multiplies inside the loop we do one. (The F[ ] array is often called the "filter kernel" or "impulse response" in the literature.) Thus, the program now looks like:

```
H = 0;
for (i=0;i<256;i=i+1)
  {
      H = H + F(600Hz)[i];     // F contains 600 and 650 Hz.
  }
```

Further, F[ ] is simply the sum of all the B[ ] arrays of all the signals we want to detect. If we want to detect 600 and 650 and 700 and 750 we simply add the B(600 Hz)[ ], B(650 Hz)[ ], B(700 Hz)[ ] and B(750 Hz)[ ] arrays. Since the B[ ] arrays are actually generated in software the entire F[ ] array can also be generated in software. This means that any filter shape can be implemented using the simple correlation routine shown above. Thus, all filters will take the same amount of time to process. This is very, very cool!

Oh and wait—remember all those weird filter cases we wanted, say notching or boosting? Well, we just take that into account when generating F[ ] and we're done. Really. Almost everything else you'll need to know about a basic DSP filtering is an optimization on how to create F[ ] or perform the correlation faster or make the correlation more precise. Here's an example…

Fourier Transform

Now let's think about the dreaded Fourier Transform. Early on in this project my eyes glazed over and I thought, "Yah Yah, but tell me how to do this stuff without Fourier Transforms, PLEASE!" After all, all the books on the subject are filled with integrals and talk about the Fourier Transform and how to implement it and how to make it fast and how clever the mathematicians were… and they were clever.

But the engineers who design cars are very clever and most drivers are not engineers. he same is true with Fourier Transforms; you can drive this car too. Early on I avoided the whole Fourier Transform view but I found that I was designing filters by formula. As I said before, we hobbyists have time to ponder and I was determined to break through the Fourier Transform barrier. I hope to help you get through this barrier as well.

The first step in breaching the Fourier Transform barrier is to simply accept a few realities.

First, just as with correlation, the Fourier Transform works on blocks of data. Accumulating, processing and delivering these blocks of data require a certain amount of logistical overhead in the programming. These are simply facts of life.

The second reality: generally speaking, larger blocks of data produce more precise results. This means that more precise results require more memory for the blocks of data and longer delays through your system. There are ways to reduce these delays but the facts remain; you want better results you're going to have to wait.

The third reality: you can probably write the code to do the Discrete Fourier Transform and it is a fascinating exercise I heartily recommend. But, you'll probably want to use a software library where someone spent lots of time making it as fast as possible. Don't get me wrong, there are many Fourier Transform applications which do not require blazing speed and I'll show you a couple later on. For now, though, let's just assume we have the appropriate software packages.

Now to simplify the discussion, I'd like to eliminate a few variables. First, let's assume all blocks of data are 100 samples long. Further, let's assume the CODEC sample rate is 100 Hz. Once we're comfortable with this simplified situation we can simply scale things. This is much like how one designs analog filters for an " =1" frequency and then scales up to the desired frequencies.

To get things started we gather up 100 CODEC samples and place them in an Input array we'll call I[ ]. The Digital Fourier Transform (DFT) then takes I[ ] and produces a second block of data which is also 100 entries long. This second block of data is, in effect, the "Spectrum" of the input block and we will call the second array the "S[ ]." The first element in this array is the "DC" value of the CODEC samples. The second element is the magnitude of 1 Hz in the signal. The third represents the magnitude of 2 Hz, the fourth 3 Hz, the fifth 4 Hz, etc. Importantly, the Spectrum Array (S[ ]) contains both upper and lower sideband spectra but let's ignore that for the moment. Authors better than I can explain these details. For now, let's just think about the lower half of the S[ ] and assume the upper half is always zero or we will MAKE it zero.

But how does each element S[i] get generated? Well, it is generated using correlation. Yep. For each "i" in S[ ] we compute S[i] = Correlation(I[ ],B(iHz)[ ]). Really, that is it. The DFT is just bunch of correlations. All the math, all the pretty graphs, all the shorthand descriptions, all the chapters on algorithms are descriptions of this simple idea: the DFT is simply a convenient collection of correlations. Don't be intimidated and don't get lost. All that math is there to help us write programs, not necessarily to help us understand.

Unfortunately, the math does tell us there is one more compli-

cation. If you think back to the very first correlation we did you'll remember that there was a point at which the correlation was zero even though the input array A[ ] and the reference array B[ ] were very similar, specifically when the two waveforms were 90 degrees out of phase. This was fine for our correlation filter but represents a problem with the DFT. Surely the I[ ] contains a signal of frequency "i." We will want S[i] to reflect that fact REGARDLESS of the phase!

The DFT avoids this problem by performing the correlation with both a sine wave and a cosine wave for each frequency. Then, independent of the phase of the signal, the appropriate DFT bin will not be zero. The DFT records the correlation for the cosine and sine waves by making each S[ ] element a complex number. The "real"part of the S[ ] element represents the correlation with the cosine and the "imaginary" part is the correlation with the sine wave. Now, just to keep things simple, we'll ignore the fact that the S[ ] elements are complex numbers. It is vastly important but let's just forget it for the time being.

Once we have the S[ ] we can process each element in that array as we see fit. For example, if you want a really narrow filter at, say, 6 Hz then you would simply zero out all S[ ] elements EXCEPT 6. You have now implemented a very narrow filter.

Or consider that you might want to notch out a 24 Hz carrier which is giving you trouble. Then you would simply set S[24] element to zero. In C you would write "S[24]=0;." Done.

For simple filters we need only need to scale each of the S[ ] array elements. When doing just this scaling, we can think of each S[i] having an associated weight or W[i]. Then we design our filter by assigning values to W[i]. To process S[ ] we simply say:

```
for (i=0;i<100;i=i+1)
  Sp[i] = S[i] * W[i];    //compute a new array called Sp[ ]
```

We would do this so that a program could write the values of W[ ] when the filter parameters changed. Then scaling S[ ] using W[ ] could be done very quickly.

There are many more advanced things you can do with the S[ ] array. I mention a few here just to pique your interest:

- You can convert frequencies. If you'd like to move the signal at 24 Hz down to 6 Hz you could simply write "S[6] = S[24];".
- You can listen to two frequencies at the same time by ADDING the Spectrum elements together. To listen to 6 Hz and 24 Hz translated down to 6 Hz you just write "S[6] = S[6] + S[24];".
- Notice that W[ ] is essentially a graphic equalizer.
- You can see how the typical spectrum display on commercial rigs is easily implemented using the S[ ] array. And the waterfall display is simply a different display program of the S[ ] array.
- You can implement "automatic fine tuning" by examining the S[ ] array and adjusting the frequency to make a chosen frequency S[chosen] the maximum.

Now, yes, I've simplified things a bit, but you can see the possibilities. There are lots of refinements to be made in any DSP approach I'll describe here. These refinements can make tremendous improvements, but they are only refinements. Once you get the basic stuff you will have the context to read the literature and understand all the refinements. Warning: "refining" can become addictive.

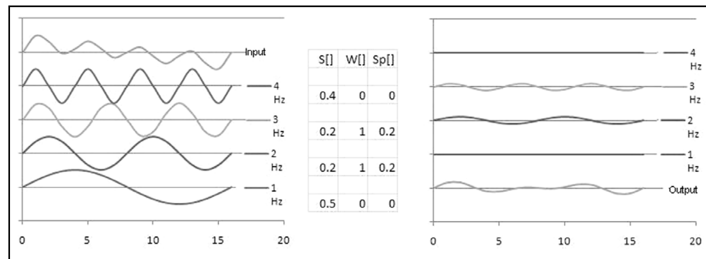

Figure 1—Overview of the software filtering process.

OK. So once we process the S[ ] array as needed we have a new array called Sp[ ]. We can't send Sp[ ] out to the headphones or speakers because the CODEC doesn't understand frequencies, it wants time samples. So we need to convert the Sp[ ] array back into a bunch of CODEC samples. This is done by using the "inverse Discrete Fourier Transform" or IDFT. Let's call the output of the IDFT the N[ ] array.

Now don't think of the IDFT as anything magical either. Think about an individual element in Sp[i]. If Sp[i] is big (say 1) then there the signal contains a lot of the frequency associated with Sp[i]. This means that N[ ] should contain a sine wave of frequency i Hz. If Sp[i] is small then N[ ] should contain a small amount of the associated sine wave. Remember, we had a way to describe this sine wave array: B(iHz)[ ]. This means that N[ ] is the sum of all the sine waves B(i)[ ] weighted by S[i]. In reality, each element N[j] is the convolution of S[i]xB(i)[j]. Look closely at that equation and try to decipher it. Said differently, "The IDFT is just the weighted sum of a bunch of sine waves."

Figure 1 shows the whole process from I[ ] to N[ ]. It is a little involved. On the left I show an input signal I[ ] at the top and four sine waves below it. Each one of these sine waves is correlated to the input generating the 4 element vertical array S[ ]. I then multiply each S[i] with the associated W[i] to implement the filter. Here I decided to implement a band pass filter which would pass just 2 Hz and 3 Hz. Having created Sp[ ] I then show the weighted sign waves on the right. Note that the top and bottom reference sine waves are straight lines; I filtered them out, remember? Then I add those sine waves together to get N[ ].

Now we have a complete path:

1) Take in a bunch of CODEC samples to make I[ ]
2) Perform the DFT on I[ ] to make S[ ]
3) Process S[ ] by weighting it using W[ ] to generate Sp[ ]
4) Perform the IDFT on Sp[ ] to create N[ ]
5) Deliver N[ ] to the CODEC.

Once you are comfortable with all this out you can start scaling the frequencies. For example, we simplified things by sampling things at 100 Hz. If our sample rate is really 16000 Hz then the bandwidth represented by S[i] is really S[i*160]. More exactly, in general the frequency in any S[i] is really i*SampleRate/BlockSize.

And there is one more important fact I need to make. All along

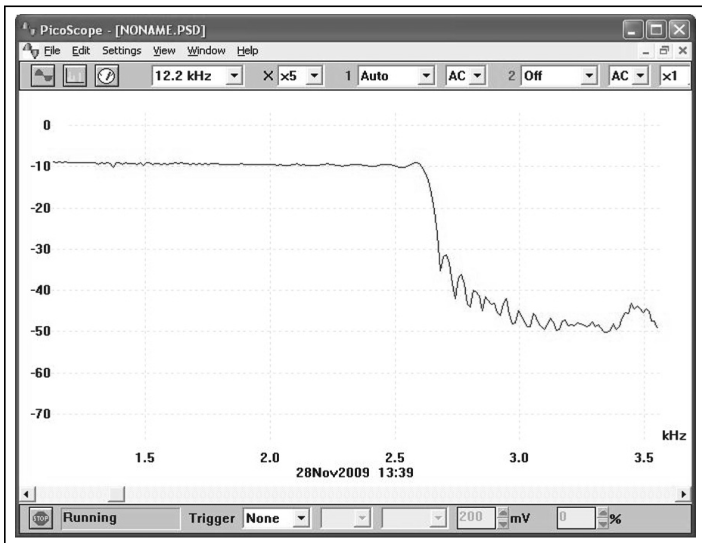

Figure 2—Filter passband with sidelobes.

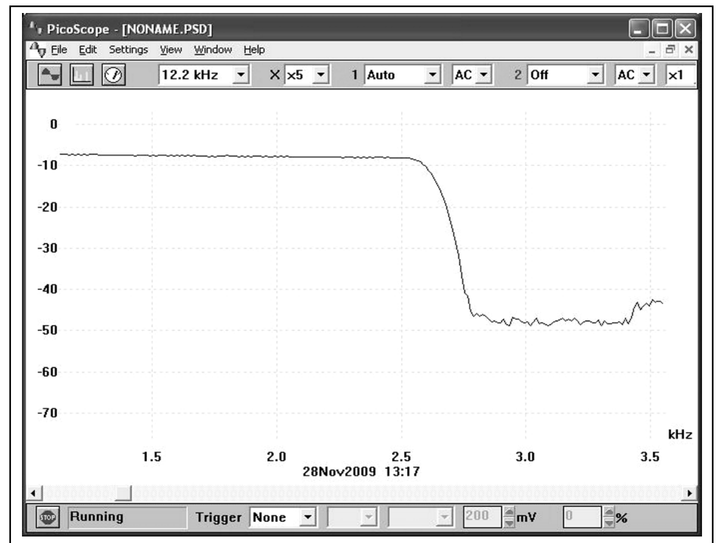

Figure 3—Filter passband after windowing is applied.

I've been writing about the Discrete Fourier Transform but most people don't talk about the DFT. Why is this? Well, if you use a block size which is a power of two then there is a really fast way to compute the DFT and it is called the Fast Fourier Transform (FFT). In general, the amount of time it takes to compute the DFT is MxN where M is the length of the correlations and N is the number of bins in the DFT. If M and N are the same size and a power of two then the FFT can compute the DFT in NlogN time. Please note though, the FFT is exactly equal to the DFT of the same data.

Wrapping Up the Fundamentals

So let's sum things up. Please remember that I'm talking informally here, all the nitty-gritty details are explained in countless publications. We're just trying to understand what is happening so we can read the books already knowing the lay of the land.

We started out thinking about how to compare an incoming signal to a reference signal and found that correlation could be used to make that comparison. Further, we saw how that comparison could be used to create a filter for a given frequency. Then we saw how a bunch of these single frequency correlation filters could be combined and a bandpass filter could be implemented. We saw that the computation of bandpass filters was no more time consuming than a filter for a single frequency because we could build a filter kernel we called F[ ] which produced any band pass we wanted.

Having understood how correlation could be used to compare signals we then turned to the Discrete Fourier Transform. We saw that the DFT was really just the result of performing a large number of correlations. The reference signals for the correlations are evenly spaced harmonics of sine and cosine waves. We called these correlations S[ ]. Having created all these S[i] elements we saw how they could be manipulated and that this manipulation was really taking place "in the frequency domain." Having processed S[ ] to create a new Sp[ ] we then saw how to convert our signal back into the "time domain" array N[ ] by summing up a bunch of weighted reference signals B(i)[ ] by using the IDFT.

A Weakness of the DFT (and Correlation as it turns out)

Up until this point all the signals we have been discussing have been selected to deliver "perfect" results. Specifically, I have chosen signals consisting of sine waves which were an even multiple of the sample period. In the DFT discussion we worked with signals that were 1 Hz, 2 Hz, 3 Hz, etc. But real world signals are not so cooperative. What happens when, say, we have a 1.25 Hz signal? Where does a 1.25 Hz signal appear in the DFT? One would hope that it would appear mostly in the 1 Hz bin and partly in the 2 Hz bin and this is mostly true; the biggest values in S[] will be in 1 Hz and 2 Hz.

Unfortunately, the complete answer is that the 1.25 Hz signal will appear, to some extent, in ALL the DFT bins. This phenomenon is called "Spectral Leakage" and is covered at length in countless publications. I have tried to find an intuitive explanation of this phenomenon but have yet to find one I like. I'll describe my best present thinking. Before looking at the solution though, let's look at the problem. Figure 2 shows what happens when I design a filter ignoring the spectral leakage problem. Examine the area around the upper frequency roll-off. Not very good, right? There are ripples in the pass band and side lobes in the stop band.

One way I've come to think about these ripples and side lobes is to consider them as simply "ringing" except here the ringing is in the "frequency domain." Consider the array W[ ] we used to scale the S[ ] array in the Spectrum. For a low pass filter, this array is a series of 1s followed by a series of zeros. If you think of this as a normal signal and run it through a circuit, you would expect some ringing as a result of the very fast falling edge. This ringing is what we are seeing in the side lobes in Figure 2!

One way to eliminate ringing on signals is to low pass filter them. Indeed, if I simply low pass filter W[ ] the side lobes will be significantly reduced. Of course, when I low pass filter W[ ] I expect the edges of the frequency response to slow down as well. In other words, I expect the roll-off to be less steep. All this can be seen in Figure 3.

The filtering of W[ ] is called Windowing and is the subject of endless debates if not downright arguments. There are many different window functions and each one has advantages and disad-

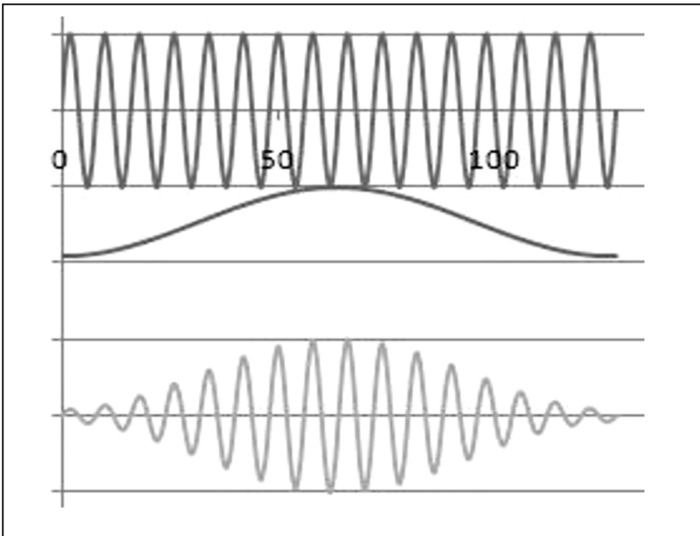

Figure 4—The Hamming window function.

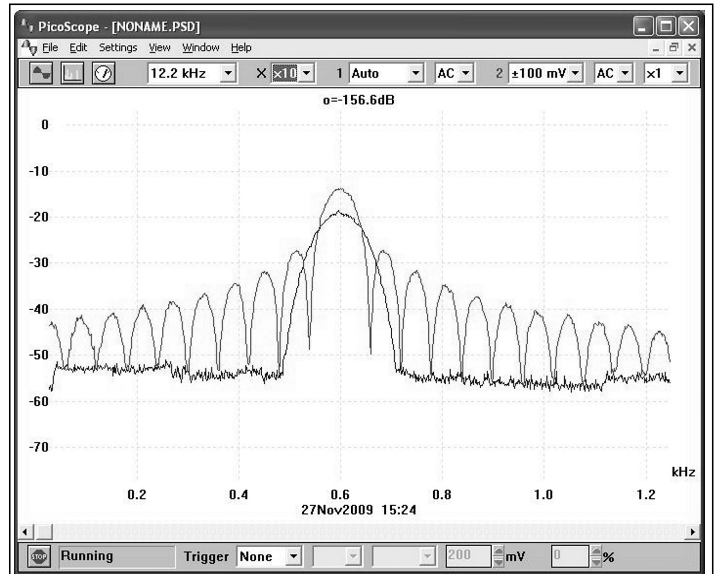

Figure 5—Comparable filters with and without windowing.

vantages. There is a general trend, though. The less you filter W[ ] the faster the roll-off will be and the larger the side lobes will be. On the other hand, the more you filter W[ ] the slower the roll-off and the smaller the side lobes. For general-purpose use I use what is called the "Hamming" window. How do I apply this window?

Well, as it turns out, most window functions are not applied to the W[ ] array at all. Rather, they are applied to the F[ ] array being used for the correlation filter. This is simply a matter of convenience, after all, there is no real difference between the F[ ] array and the W[ ] array. Indeed, later we'll see how to convert between the two. OK, so the windowing is done in the F[ ] array. Just how? Well, in the case of the Hamming window the F[ ] array is multiplied by $(0.54-0.46\cos(x))$. Figure 4 shows this graphically for a single frequency filter. Notice how the values of the final F[ ] array (shown at the bottom) start small, grow and then shrink again.

Figure 5 compares two filters, one with the Hamming filter and one without. Both filters are exactly 1 DFT bin wide. The wide trace with no significant side lobes is a single bin with the Hamming window applied. The taller and narrower trace is an unwindowed version of the filter. Notice the roll off is faster but the side lobes are much larger in the unwindowed case.

Let me close this section with a couple of remarks. In this description of correlation and DFTs, I have glossed over many details some of which are critical. My goal was not to provide a cookbook of how to use correlation and the DFT/IDFT pair. Rather, I hope I have shown you that these technologies are not mystical and can be understood intuitively with only a little guidance. I hope that I have laid a foundation of understanding that gets you over the initial hurdle of using this powerful technology.

Now let's explore a few of the fundamental digital signal processing techniques used in my SDR. It is time to show how some of this stuff gets used in my SDR. Here are a few notes on various problems I encountered and how I solved them.

The ejm Function

One of the very first subroutines I had to write was a fast version of "sin" and "cos." The dsPIC C library provides these functions but they are designed as general purpose and precise functions which return high precision numbers. These subroutines take hundreds of microseconds or even milliseconds to complete; much too long for real time DSP operation.

There are several ways to implement fast versions of these functions with varying precision/speed tradeoffs. I ended up choosing a simple table lookup scheme. My table is 1024 entries long and does interpolation between the points. I call my subroutine "ejm." "ejm" takes a single Fractional argument which is the angle and returns a Complex number. The precise code is a straight forward exercise in programming and not shown here. The following code is shown for clarification. (Take Note: This is NOT functional code, I have left out details concerning the translation of Fractional to C language "floats.")

```
// function taking one Fractional
// and returning Complex

Complex ejm(Fractional angle)
  {
     Complex rval;
     rval.i = cos(PI*angle);
     rval.q = –sin(PI*angle);
     return(rval);
  }
```

Note that because the "angle" argument is a Fractional it can have values from (essentially) –1 through 1. Note also that this means that we don't need to worry about the angle ever being outside the range of –PI to PI. This eliminates any bounds checking which might need to be performed.

This routine is used extensively throughout my code whenever a sine or cosine wave is needed.

The Hilbert Transform

I chose to implement the Hilbert Transform using an FIR-like construct. (FIRs are a method of implementing filters which, for

the purposes of this discussion, need not be further described.) Indeed, the only difference between a typical FIR filter and the Hilbert Transform is the coefficients in the F[ ] array. The coefficients in the F[ ] array can be computed using the IDFT or they can be computed in a closed form. I chose to compute the Hilbert Transform coefficients using the equations found in EMFRD.

There are other ways to generate the Hilbert Transform. A literature search is worthwhile. There doesn't seem to be any one way which is significantly better than all the others so I didn't pursue their implementation.

As with normal FIR filters the Hilbert Transform benefits from a windowing function applied to the F[ ] array. I use the Blackman-Harris window when implementing the Hilbert Transform.

General Purpose Filters

In the present incarnation of my SDR, I use correlation to implement my general purpose filtering. To do so, I generate my F[ ] array on the fly. I provide for three parameters in my user interface: lower cut-off frequency, upper cut-off frequency, and "filter length." To generate my F[ ] array I use a technique called "Windowed Sync." The code was taken almost literally from Reference 3. The algorithm is not very interesting but it is very instructive to play with each of the parameters.

Of particular interest is the "filter length" parameter. As discussed above, better filters require longer block lengths. As one shortens the F[ ] length the edges of the filter will get slower and the stop band attenuation will get worse. Conversely, as one increases the length the edges get steeper and the stop band attenuation gets better…. up to a point. In my SDR I found that when increasing the block length past 250 points, my filters simply did not improve. I'll return to this topic shortly.

There is little doubt that there are algorithms that produce "better" filters than the simple Windowed Sync—better in terms of roll off, ripple, stop band attenuation, FIR filter length, etc. However, the basic Windowed Sync produces very good results, is easy to implement and provides a transparent technology suitable for experimentation.

Narrow Filters

In researching the topic of filtering, I came to understand why my longer FIR filters did not produce better results. When doing very long convolution, the F[ ] array element values get quite small. I found this by looking closely at the F[ ] array values and then the algorithm itself. The center of the algorithm in Reference 3 is an equation which has the form sin(ki)/i where i is the index in the F[ ] array. This means that as the F[ ] array gets larger the i gets larger and the "/i" piece of the equation gets me in trouble.

An additional piece of the puzzle comes from the windowing function. Essentially all windowing functions will have this problem; it is not peculiar to mine. The key point of the windowing function is the term that looks like "1–cos(x)" so when x is getting close to the ends of the F[ ] array, the "1–cos( )" term gets very close to zero.

The combination of these two features makes the F[ ] array vanish below 16 bit resolution. I thought long and hard about how to get around this problem. Ultimately, I found a technology called "Sliding DFT" with frequency domain windowing. As it turns out, I did not implement the Sliding DFT exactly. I have seen my modification called the "Running DFT," a term that is a little less formal so I'll go with that name. Let's start with how to implement the Running DFT.

The Running DFT allows me to compute the value of any single bin in the DFT one CODEC sample at a time. As with the filters we've discussed earlier, the basic technique is essentially a running correlation of the incoming data with a reference signal. The first step is to implement the reference signal. Let's continue our discussion assuming I would like to correlate 1024 samples of the incoming signal.

Before delving into the code there are a couple of important concepts that need pointing out. First, I need to avoid working with small numbers whenever possible and I certainly don't want to do any division (too time consuming). Second, I want the calculation to be "stable." It is very important that intermediate numbers not grow. As a result, I need to worry about when precision is being lost. Now, on with the discussion…

Remember that the correlation is the sum of the product of the incoming signal and the reference signal. Each time a new CODEC sample arrives I need to compute the product and add it to the sum. In addition, I need to subtract out the contribution of the 1024th oldest input. For this discussion I use a ring to keep the 1024 entries; I won't show the ring code again.

In order to allow me to write more concise programs (and therefore programs with fewer bugs) I'm going to keep the "sums" as BigComplex numbers; see the declaration below. Note that I already introduced two functions for manipulating Complex numbers: CxC to multiply and CpC to add. These were necessary because C does not have a "native" Complex number type. For the purposes of the code below, however, I'm going to use * and + to indicate multiplication and addition. The real code would need to use CxC and CpC.

Here is the basic code for implementing a Sliding DFT for bin number "i":

```
typedef struct {long i; long q;} BigComplex;
BigComplex aSumi;
Complex tempC;

oldestI = inputRing[ringIndex];
inputRing[ringIndex] = I;

tempC = ejm(i*ringIndex/1024);
aSumi = aSumi + tempC * I;          //correlate and add
                                      in new.
aSumi = aSumi - tempC * oldestI;    //take out old.
```

There are a couple of important details I slipped into the above code. First, the reference signals make complete and exact cycles every 1024 samples. The inputRing also cycles every 1024 cycles. Thus, I guarantee that the reference signals have EXACTLY the same value when a CODEC sample is "added in" and when it is "subtracted out" 1024 cycles later. Second, the running sums are 32 bits long so there is no chance of an overflow. The maximum value which might be in "aSum"is only 512. Third, notice that I add in and subtract out values directly to the sum. This is because when using Fractionals there is no absolute assurance that

"a(b+c)" is exactly the same as "ab + ac." Rounding can make a difference.

Believe it or not, that is all one needs to compute the running DFT. However, we still need to apply a windowing function or there will be significant side lobes. Let's explore how to apply a window in the frequency domain using the Hamming Window as an example. Remember that the Hamming Window equation was 0.54–0.46 cos(x).

This Hamming Window equation is written "in the time domain," and we want to perform the windowing function "in the frequency domain." How do we convert from "time" to "frequency"? Yep, we apply the DFT to the equation 0.54-0.46 cos(x) window. A little bit of examination of the equation is quite telling. The first term of the equation is 0.54. This represents a DC value and we would expect the DFT of this equation to have a first bin S[ ] equal to 0.54. Second, the "cos" part cycles exactly once during the window and so we would expect the second bin s[1] to be of value 0.46. Well, we are half right. As it turns out, the DFT of the Hamming window has exactly 3 non-zero bins. S[0] is 0.54 as we expect but the .46 part is divided equally between the S[1] and S[-1] bins; each is –0.23. Generally speaking, real cosine signals always have spectra which are symmetric around 0. This is something you'll see repeatedly.

Now don't get flustered by that "-1," it is just math and it will go away in just a moment. We need to shift this window function up to the filter frequency of interest, here bin "i." The final W[] array will have three nonzero bins: W[i–1]=–0.23, W[i]=0.54 and W[i+1] is –0.23. Done. We've applied the Hamming Window to our single bin DFT. Unfortunately, this means we need to compute the running DFT of three bins rather than just one but that is a simple extension of the above code.

We have now computed the Sp[i-1],Sp[i] & Sp[i+1] bins but we aren't done, we need to compute the running IDFT. Fortunately, this is pretty easy too. To compute the running IDFT of a single bin we need to simply multiply the Sp[i] bins by the appropriate reference signals. I will again use * and + so as to improve readability, but the real code would need to use CxC and CpC as before. Also, please take note that the angle arguments to ejm are negated; the IDFT reference signals run in the reverse direction of the DFT references.

```
N =  –0.23 * ejm(–(i–1)*ringIndex/1024);
N = N + 0.54 * ejm(–i*ringIndex/1024);
N = N – 0.23 * ejm(–(i+1)*ringIndex/1024);
```

The final step is to scale the final value N so that it isn't too big. It turns out that any single DFT bin can reach the value equal to the length. In this case that is 1024. Thus, the final N above can be as large as 1024 (1024*0.54 + 1024*0.46). Fortunately, this is a power of two and a simple shift wraps up the computation. N can now be handed to the CODEC for output.

The running DFT allows you to build very tight filters without doing any division (well, almost) and without losing any precision. If one makes a few optimizations the above code can run VERY fast. The key optimization is to replace a call to ejm with a direct table lookup. This is possible because it turns out that there is no need for interpolation if the table is large enough. Other optimizations are performed in the final version of the code but these optimizations tend to make the code less readable and so are not shown here.

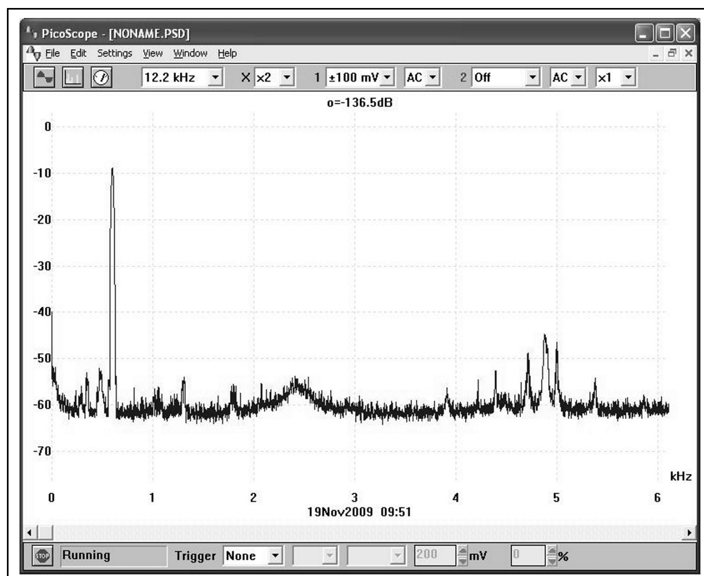

Figure 6—A 600 Hz filter using the sliding DFT process.

Figure 6 shows a filter for 600 Hz using the Sliding DFT. This filter rolls off over 43 dB in just 50 Hz. Using a slightly more aggressive Window (Blackman-Harris) this roll-off can be 73 dB in the same 50 Hz!

One final note: if you were particularly observant you would have noticed that this filter uses a complex input I. When this is done then the Running DFT does sideband selection in addition to filtering. Thus, when using the Running DFT as a filter as I do here, there is no need for the Hilbert Transform.

Arbitrary Filters

We spoke at length about how to implement arbitrary filters using both the DFT and correlation. It was fairly simple to see how to implement funky filters in the DFT, we simply had to scale each element of the S[ ] array. Remember that we used a second array W[ ] to express these scale factors. Then we would perform a DFT, scale the spectrum using W[ ] and then perform an IDFT. So designing our filter in the frequency domain was pretty easy: just assign the appropriate values to W[ ].

However, in discussing correlation I conveniently glossed over how to create an arbitrary filter array F[ ]. As it turns out, you already have all the tools necessary. Remember that we wanted to create F[ ] by adding a bunch of reference signals together? Well, that is exactly what the IDFT does, right? So to create F[ ] we simply perform the IDFT of the W[ ] array and, *voila*, an F[ ] suitable for use in Convolution! But don't forget to apply a window!

There are, of course, many other ways to generate the F[ ] array. Some of these methods generate better results; faster edges, shorter F[ ] arrays, flatter pass bands or deeper stop bands. Most of these tools cost money and are largely opaque. A professional engineer would benefit from their use. The vast majority of hobbyists will find this simple approach more than adequate.

Using the FFT in the Data Path

Think back about the description of the DFT and IDFT. In that

section I described a way to perform filtering completely in the frequency domain. The basic approach was described as I[ ] –> DFT –> S[ ] –> IDFT –> N[ ]; input converted to spectrum, processed and then converted back to output. This approach to signal processing is called “Fast Convolution” and there are some complications involved. A little research is in order if you are interested.

I have tried numerous times to make this work and have failed each and every time. I have coded the approach independently several times using several different algorithms and always failed to achieve acceptable performance; specifically, every attempt has produced a low frequency “beat” tone that was related to the size of the input ring. I have tried to understand why this has not worked and would appreciate any insight anyone might offer.

My best understanding of this problem right now is that 16 bit, fixed point arithmetic simply loses too much precision during the calculation of the FFT and IFFT. I am still intrigued by this approach and keep trying. The next effort will be to code up my own highly optimized version of the FFT and see what happens when I increase the precision of the arithmetic.

I would point out that I don’t believe that there is really that much to be gained by using this approach in my radio. For basic filters either correlation or the Running DFT can perform adequately and are much easier to program and maintain.

Summary

With this portion of my article, I have completed the description of the SDR’s CW and SSB receiver. I have not worked on a SSB exciter as my Class E power amplifier will not support SSB.

My Software Defined Radio project has been and continues to be a grand adventure. It has afforded me countless hours of education, entertainment, distraction, frustration and excitement… and I treasure every minute. In this series of articles, I have discussed a variety of obstacles I have encountered and solutions I have employed. Most of these obstacles were typical problems with typical solutions. When a problem was familiar I endeavored to use the “standard” solution. Other problems were new to me and required significant research before the problem was totally understood and a solution was selected.

My entire design database; schematics, pc board layout, parts list and software will be provided on the *QRP Quarterly* Web site as well as my own which is www.ae6ty.com. This will allow you to see exactly how I solved (or ignored or overlooked) a problem.

Through this article, I hope I have piqued your interest in building and perhaps even designing your own equipment. At the very least, I hope I have given you a glimpse of some of the techniques and capabilities of the emerging technology called Software Defined Radio.

In the next issue of *QRP Quarterly* we’ll shift gears to a more hardware oriented portion of the project; the output power amplifier. That article, Part IV of the series, will explore the design methodology and computer tools used to implement a single band, 5 watt, class E amplifier.

—73/72, Ward, AE6TY

Bibliography

1. Lyons, Richard G., *Understanding Digital Signal Processing*, Second Edition, Prentice Hall, Upper Saddle River, New Jersey 2004

2. Hayward, Campbell, Larkin, *Experimental Methods in RF Design*, The American Radio Relay League, 2003

3. Smith, Steven W., *Digital Signal Processing A Practical Guide for Engineers and Scientists*, Newnes, 2003

One More Note from WA8MCQ’s Idea Exchange...

Cleaning Up NiCd Battery Leakage

Gary McCaughey, W2UX posted this on qrp-l@qth.net—

A nice StreamLite flash light with a nickel cadmium (NiCd) battery stick inside was just given to me. I opened it and there is white dust from the battery inside. You can see a spot where there is some sort of corrosion. What can you use to clean this stuff? It is not like a lead acid battery that you can clean up with baking soda. Also, this stuff is considered toxic, right?

This reply came from Brad Thompson, AA1IP—

If I recall correctly, the white residue is sodium hydroxide electrolyte (an alkali or base), which you can remove with a discarded toothbrush or a cotton swab dipped in a weak acid such as vinegar. Once the residue is removed, swab the area with distilled water (or tap water) to remove any chemical residue and allow it to dry. Inspect for damage—if the battery contact region isn’t too badly damaged, use as is. Otherwise, you may have to improvise a replacement contact.

DE WA8MCQ—

I checked Wikipedia, the online encyclopedia, and it indicated that the electrolyte is potassium hydroxide (KOH). Some other web sites indicated the same thing. To be safe I decided to check the web site of Saft, a company that actually makes this sort of thing. (Not exactly a household term in this country, it’s a large French company that has been around for a very long time.) It says, “Nickel-cadmium cells have an anode (negative) in cadmium hydroxide and a cathode (positive) in nickel hydroxide, immersed in an alkaline solution (electrolyte) comprising potassium, sodium and lithium hydroxides.”

I’ve had good luck cleaning up NiCd residue with vinegar, as Brad suggests. It also works well if you have a leaking alkaline battery. If you have a battery leakage situation and can’t tell what type it was (such as someone handing you a device with the leaking batteries already gone), try both vinegar and a bit of baking soda in water. One or the other should take it right off.