A Software Defined Radio for the Masses, Part 3

Learn how to use DSP to make the PC sound-card interface from Part 2 into a functional software-defined radio. We also explore a powerful filtering technique called FFT fast-convolution filtering.

By Gerald Youngblood, AC5OG

Part 1[1] of this series provided a general description of digital signal processing (DSP) as used in software-defined radios (SDRs) and included an overview of a full-featured radio that uses a PC to perform all DSP and control functions. Part 2[2] described *Visual Basic* source code that implements a full-duplex quadrature interface to a PC sound card.

As previously described, *in-phase (I)* and *quadrature (Q)* signals give the ability to modulate or demodulate virtually any type of signal. The Tayloe Detector, described in Part 1, is a simple method of converting a modulated RF signal to baseband in quadrature, so that it can be presented to the left and right inputs of a stereo PC sound card for signal processing. The full-duplex DirectX8 interface, described in Part 2, accomplishes the input and output of the sampled quadrature signals. The sound-card interface provides an input buffer array, *inBuffer*(), and an output buffer array, *outBuffer*(), through which the

[1]Notes appear on page 00.

8900 Marybank Dr
Austin, TX 78750
AC5OG@arrl.org

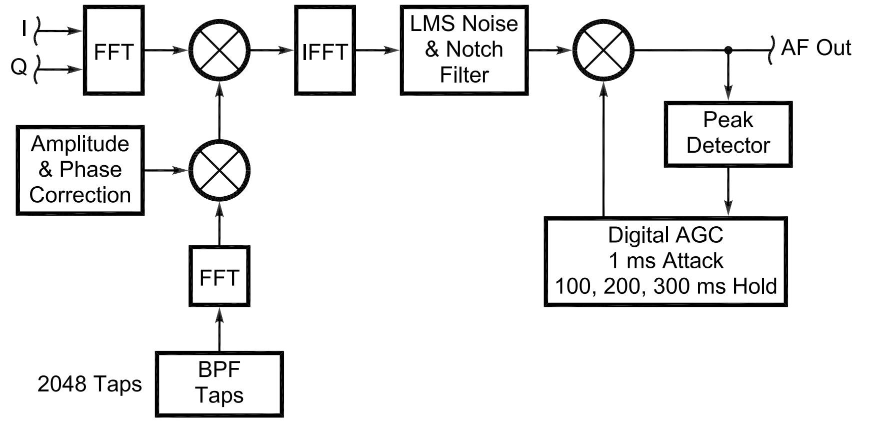
I
Q
FFT
IFFT
LMS Noise & Notch Filter
AF Out
Peak Detector
Amplitude & Phase Correction
Digital AGC 1 ms Attack 100, 200, 300 ms Hold
FFT
2048 Taps
BPF Taps

Fig 1—DSP software architecture block diagram.

DSP code receives the captured signal and then outputs the processed signal data.

This article extends the sound-card interface to a functional SDR receiver demonstration. To accomplish this, the following functions are implemented in software:

- Split the stereo sound buffers into *I* and *Q* channels.
- Conversion from the time domain into the frequency domain using a *fast Fourier transform (*FFT*)*.
- Cartesian-to-polar conversion of the signal vectors.
- Frequency translation from the 11.5 kHz-offset baseband IF to 0 Hz.
- Sideband selection.
- Band-pass filter coefficient generation.
- FFT fast-convolution filtering.
- Conversion back to the time domain with an inverse fast Fourier transform (IFFT).
- Digital automatic gain control (AGC) with variable hang time.
- Transfer of the processed signal to the output buffer for transmit or receive operation.

The demonstration source code may be downloaded from *ARRLWeb*.[3] The software requires the *dynamic link library (DLL)* files from the Intel

```
Public Const Fs As Long = 44100 'Sampling frequency in samples per second
Public Const NFFT As Long = 4096          'Number of FFT bins
Public Const BLKSIZE As Long = 2048       'Number of samples in capture/play block
Public Const CAPTURESIZE As Long = 4096          'Number of samples in Capture Buffer
Public Const FILTERTAPS As Long = 2048           'Number of taps in bandpass filter
Private BinSize As Single'Size of FFT Bins in Hz

Private order As Long      'Calculate Order power of 2 from NFFT
Private filterM(NFFT) As Double   'Polar Magnitude of filter freq resp
Private filterP(NFFT) As Double   'Polar Phase of filter freq resp
Private RealIn(NFFT) As Double    'FFT buffers
Private RealOut(NFFT) As Double
Private ImagIn(NFFT) As Double
Private ImagOut(NFFT) As Double

Private IOverlap(NFFT - FILTERTAPS - 1) As Double          'Overlap prev FFT/IFFT
Private QOverlap(NFFT - FILTERTAPS - 1) As Double          'Overlap prev FFT/IFFT

Private RealOut_1(NFFT) As Double         'Fast Convolution Filter buffers
Private RealOut_2(NFFT) As Double
Private ImagOut_1(NFFT) As Double
Private ImagOut_2(NFFT) As Double

Public FHigh As Long       'High frequency cutoff in Hz
Public FLow As Long        'Low frequency cutoff in Hz
Public Fl As Double        'Low frequency cutoff as fraction of Fs
Public Fh As Double        'High frequency cutoff as fraction of Fs
Public SSB As Boolean      'True for Single Sideband Modes
Public USB As Boolean      'Sideband select variable
Public TX As Boolean       'Transmit mode selected
Public IFShift As Boolean'True for 11.025KHz IF

Public AGC As Boolean      'AGC enabled
Public AGCHang As Long            'AGC AGCHang time factor
Public AGCMode As Long            'Saves the AGC Mode selection
Public RXHang As Long      'Save RX Hang time setting
Public AGCLoop As Long'AGC AGCHang time buffer counter
Private Vpk As Double      'Peak filtered output signal
Private G(24) As Double    'Gain AGCHang time buffer
Private Gain As Double     'Gain state setting for AGC
Private PrevGain As Double        'AGC Gain during previous input block
Private GainStep As Double        'AGC attack time steps
Private GainDB As Double          'AGC Gain in dB
Private TempOut(BLKSIZE) As Double        'Temp buffer to compute Gain
Public MaxGain As Long     'Maximum AGC Gain factor

Private FFTBins As Long 'Number of FFT Bins for Display
Private M(NFFT) As Double         'Double precision polar magnitude
Private P(NFFT) As Double         'Double precision phase angle
Private S As Long          'Loop counter for samples
```

Figure 2 – Variable Declarations

Signal Processing Library[4] to be located in the working directory. These files are included with the demo software.

The Software Architecture

Fig 1 provides a block diagram of the DSP software architecture. The architecture works equally well for both transmit and receive with only a few lines of code changing between the two. While the block diagram illustrates functional modules for *Amplitude and Phase Correction* and the *LMS Noise and Notch Filter*, discussion of these features is beyond the scope of this article.

Amplitude and phase correction permits imperfections in phase and amplitude imbalance created in the analog circuitry to be corrected in the frequency domain. LMS noise and notch filters[5] are an adaptive form of *finite impulse response* (FIR) filtering that accomplishes noise reduction in the time domain. There are other techniques for noise reduction that can be accomplished in the frequency domain such as *spectral subtraction,*[6] *correlation*[7] and *FFT averaging*[8]

Parse the Input Buffers to Get *I* and *Q* Signal Vectors

Fig 2 provides the variable and constant declarations for the demonstration code. The structure of the *inBuffer*() format is illustrated in Fig 3. The left and right signal inputs must be parsed into *I* and *Q* signal channels before they are presented to the FFT input. The 16-bit integer left- and right-channel samples are interleaved, therefore the code shown in Fig 3 must be used to split the signals. The arrays *RealIn*() and *RealOut*() are used to store the *I* signal vectors and the arrays *ImagIn*() and *ImagOut*() are used to store the *Q* signal vectors. This corresponds to the nomenclature used in the complex FFT algorithm. It is not critical which of the *I* and *Q* channels goes to which input because one can simply reverse the code in Fig 3 if the sidebands are inverted.

The FFT: Conversion to the Frequency Domain

Part 1 of this series discussed how the FFT is used to convert discrete-time sampled signals from the time domain into the frequency domain (see Note 1). The FFT is quite complex to derive mathematically and somewhat tedious to code. Fortunately, Intel has provided performance-optimized code in DLL form that can be called from a single line of code for this and other important DSP functions (see Note 4).

The FFT effectively consists of a series of very narrow band-pass filters, the outputs of which are *bins,* as illustrated in Fig 4. Each bin has a magnitude and phase value representative of the sampled input signal's content at the respective bin's center frequency. Overlap of adjacent bins resembles the output of a *comb filter* as discussed in Part 1.

The PC SDR uses a 4096-bin FFT. With a sampling rate of 44,100 Hz, the bandwidth of each bin is 10.7666 Hz (44,100/4096), and the center frequency of each bin is the bin number times the bandwidth. Notice in Fig 4 that with respect to the center frequency of the sampled quadrature signal, the upper sideband is located in bins 1 through 2047, and the lower sideband is located in bins 2048 through 4095. Bin 0 contains the carrier translated to 0 Hz. An FFT performed on an analytic signal $I + jQ$ allows positive and negative frequencies to be analyzed separately.

The Turtle Beach Santa Cruz sound card I use has a 3-dB frequency response of approximately 10 Hz to 20 kHz. (Note: the data sheet states a high-frequency cutoff of 120 kHz, which has to be a typographical error, given the 48-kHz maximum sampling

```
Erase RealIn, ImagIn

For S = 0 To CAPTURESIZE - 1 Step 2     'Copy I to RealIn and Q to ImagIn
   RealIn(S \ 2) = inBuffer(S + 1)  'Zero stuffing second half of
   ImagIn(S \ 2) = inBuffer(S)             'RealIn and ImagIn
Next S
```

Figure 3 – Parsing Input Buffers Into *I* and *Q* Signal Vectors

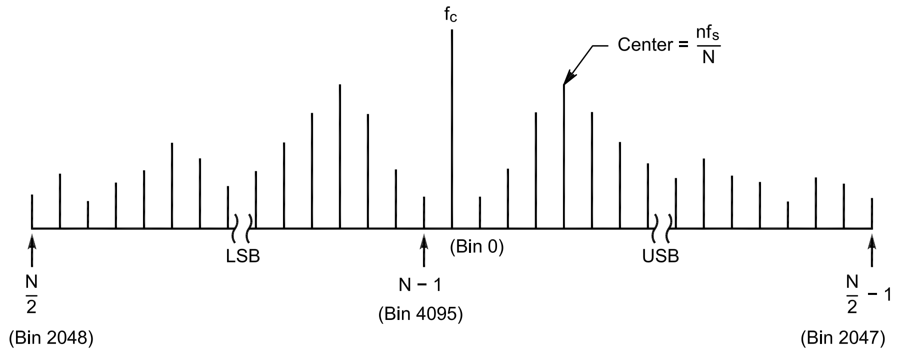

Fig 4—FFT output bins.

```
nspzrFftNip RealIn, ImagIn, RealOut, ImagOut, order, NSP_Forw
nspdbrCartToPolar RealOut, ImagOut, M, P, NFFT 'Cartesian to polar
```

Figure 5 – Time Domain to Frequency Domain Conversion Using the FFT

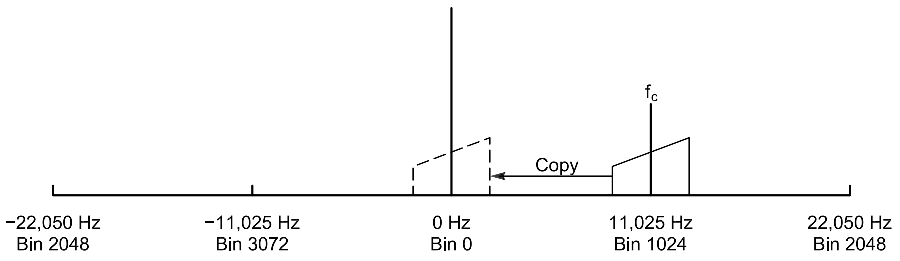

Fig 6—Offset baseband IF diagram. The local oscillator is shifted by 11.025 kHz so that the desired-signal carrier frequency is centered at an 11,025-Hz offset within the FFT output. To shift the signal for subsequent filtering the desired bins are simply copied to center the carrier frequency, f_C, at 0 Hz.

rate). Since we sample the RF signal in quadrature, the sampling rate is effectively doubled (44,100 Hz times two channels yields an 88,200-Hz effective sampling rate). This means that the output spectrum of the FFT will be twice that of a single sampled channel. In our case, the total output bandwidth of the FFT will be 10.7666 Hz times 4096 or 44,100 Hz. Since most sound cards roll off near 20 kHz, we are probably limited to a total bandwidth of approximately 40 kHz.

Fig 5 shows the DLL calls to the Intel library for the FFT and subsequent conversion of the signal vectors from the Cartesian coordinate system to the Polar coordinate system. The nspzrFftNip routine takes the time domain *RealIn*() and *ImagIn*() vectors and converts them into frequency domain *RealOut*() and *ImagOut*() vectors. The order of the FFT is computed in the routine that calculates the filter coefficients as will be discussed later. NSP_Forw is a constant that tells the routine to perform the forward FFT conversion.

In the Cartesian system the signal is represented by the magnitudes of two vectors, one on the *Real* or *x* plane and one on the *Imaginary* or *y* plane. These vectors may be converted to a single vector with a magnitude (*M*) and a phase angle (*P*) in the polar system. Depending on the specific DSP algorithm we wish to perform, one coordinate system or the other may be more efficient. I use the polar coordinate system for most of the signal processing in this example. The nspdbrCartToPolar routine converts the output of the FFT to a polar vector consisting of the magnitudes in *M*() and the phase values in *P*(). This function simultaneously performs Eqs 3 and 4 in Part 1 of this article series.

Offset Baseband IF Conversion to Zero Hertz

My original software centered the RF carrier frequency at bin 0 (0 Hz). With this implementation, one can display (and hear) the entire 44-kHz spectrum in real time. One of the problems encountered with direct-conversion or zero-IF receivers is that noise increases substantially near 0 Hz. This is caused by several mechanisms: 1/*f* noise in the active components, 60/120-Hz noise from the ac power lines, microphonic noise caused by mechanical vibration and local-oscillator phase noise. This can be a problem for weak-signal work because most people tune CW signals for a 700-1000 Hz tone. Fortunately, much of this noise disappears above 1 kHz.

Given that we have 44 kHz of spectrum to work with, we can offset the digital IF to any frequency within the FFT output range. It is simply a matter of deciding which FFT bin to designate as the carrier frequency and then offsetting the local oscillator by the appropriate amount. We then copy the respective bins for the desired sideband so that they are located at 0 Hz for subsequent processing. In the PC SDR, I have chosen to use an offset IF of 11,025 Hz, which is one fourth of the sampling rate, as shown in Fig 6.

Fig 7 provides the source code for shifting the offset IF to 0 Hz. The carrier frequency of 11,025 Hz is shifted to bin 0 and the upper sideband is shifted to bins 1 through 1023. The lower sideband is shifted to bins 3072 to 4094. The code allows the IF shift to be enabled or disabled, as is required for transmitting.

```
IFShift = True     'Force to True for the demo

  If IFShift = True Then   'Shift sidebands from 11.025KHz IF
    For S = 0 To 1023
      If USB Then
        M(S) = M(S + 1024)      'Move upper sideband to 0Hz
        P(S) = P(S + 1024)
      Else
        M(S + 3072) = M(S + 1)'Move lower sideband to 0Hz
        P(S + 3072) = P(S + 1)
      End If
    Next
  End If
```

Figure 7 –Code for Down Conversion From Offset Baseband IF to 0Hz

```
If SSB = True Then         'SSB or CW Modes
    If USB = True Then
      For S = FFTBins To NFFT - 1        'Zero out lower sideband
        M(S) = 0
      Next
    Else
      For S = 0 To FFTBins - 1  'Zero out upper sideband
        M(S) = 0
      Next
    End If
  End If
```

Figure 8 – Sideband Selection Code

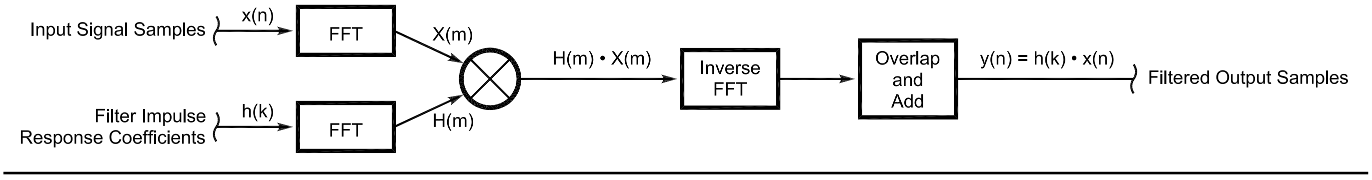

Fig 9—FFT fast-convolution-filtering block diagram. The filter impulse-response coefficients are first converted to the frequency domain using the FFT and stored for repeated use by the filter routine. Each signal block is transformed by the FFT and subsequently multiplied by the filter frequency-response magnitudes. The resulting filtered signal is transformed back into the time domain using the inverse FFT. The Overlap/Add routine corrects the signal for circular convolution.

Selecting the Sideband

So how do we select sideband? We store zeros in the bins we don't want to hear. How simple is that? If it were possible to have perfect analog amplitude and phase balance on the sampled *I* and *Q* input signals, we would have infinite sideband suppression. Since that is not possible, any imbalance will show up as an image in the passband of the receiver. Fortunately, these imbalances can be corrected through DSP code either in the time domain before the FFT or in the frequency domain after the FFT. These techniques are beyond the scope of this discussion, but I may cover them in a future article. My prototype using INA103 instrumentation amplifiers achieves approximately 40 dB of opposite sideband rejection *without* correction in software.

The code for zeroing the opposite sideband is provided in Fig 8. The lower sideband is located in the high-numbered bins and the upper sideband is located in the low-numbered bins. To save time, I only zero the number of bins contained in the *FFTBins* variable.

FFT Fast-Convolution Filtering Magic

Every DSP text I have read on single-sideband modulation and demodulation describes the IF sampling approach. In this method, the A/D converter samples the signal at an IF such as 40 kHz. The signal is then quadrature down-converted in software to baseband and filtered using finite impulse response (FIR)[9] filters. Such as system was described in Doug Smith's *QEX* article called, "Signals, Samples, and Stuff: A DSP Tutorial (Part 1)."[10] With this approach, all processing is done in the time domain.

For the PC SDR, I chose to use a very different approach called *FFT fast-convolution filtering* (also called *FFT convolution*) that performs all filtering functions in the frequency domain.[11] An FIR filter performs convolution of an input signal with a filter impulse response in the time domain. Convolution is the mathematical means of combining two signals (for example, an input signal and a filter impulse response) to form a third signal (the filtered output signal).[12] The time-domain approach works very well for a small number of filter taps. What if we want to build a very-high-performance filter with 1024 or more taps? The processing overhead of the FIR filter may become prohibitive. It turns out that an important property of the Fourier transform is that convolution in the time domain is equal to multiplication in the frequency domain. Instead of directly convolving the input signal with the windowed filter impulse response, as with a FIR filter, we take the respective FFTs of the input signal and the filter impulse response and simply multiply them together, as shown in Fig 9. To get back to the time domain, we perform the inverse FFT of the product. FFT con-

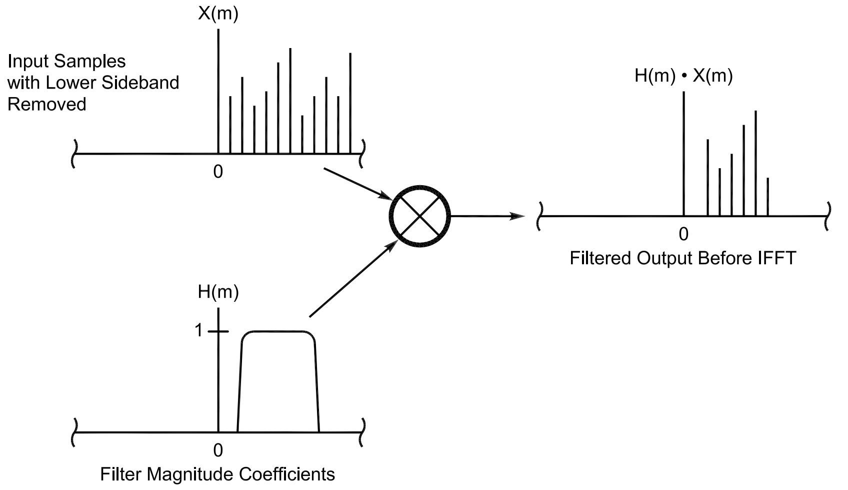
X(m)
Input Samples with Lower Sideband Removed
0
H(m) • X(m)
0
Filtered Output Before IFFT
H(m)
1
0
Filter Magnitude Coefficients

Fig 10—FFT fast convolution filtering output. When the filter-magnitude coefficients are multiplied by the signal-bin values, the resulting output bins contain values only within the pass-band of the filter.

```
Static Rh(NFFT) As Double        'Impulse response for bandpass filter
  Static Ih(NFFT) As Double      'Imaginary set to zero
  Static reH(NFFT) As Double     'Real part of filter response
  Static imH(NFFT) As Double     'Imaginary part of filter response

  Erase Ih

  Fh = FHigh / Fs          'Compute high and low cutoff
  Fl = FLow / Fs'as a fraction of Fs
  BinSize = Fs / NFFT    'Compute FFT Bin size in Hz

  FFTBins = (FHigh / BinSize) + 50          'Number of FFT Bins in filter width

  order = NFFT  'Compute order as NFFT power of 2
  Dim O As Long

  For O = 1 To 16          'Calculate the filter order
    order = order \ 2
    If order = 1 Then
      order = O
      Exit For
    End If
  Next

  'Calculate infinite impulse response bandpass filter coefficients
  'with window
  nspdFirBandpass Fl, Fh, Rh, FILTERTAPS, NSP_WinBlackmanOpt, 1

  'Compute the complex frequency domain of the bandpass filter
  nspzrFftNip Rh, Ih, reH, imH, order, NSP_Forw
  nspdbrCartToPolar reH, imH, filterM, filterP, NFFT

End Sub
```

Figure 11 – Code for the Generating Bandpass Filter Coefficients in the Frequency Domain

volution is often faster than direct convolution for filter kernels longer than 64 taps, and it produces exactly the same result.

For me, FFT convolution is easier to understand than direct convolution because I mentally visualize filters in the frequency domain. As described in Part 1 of this series, the output of the complex FFT may be thought of as a long bank of narrow band-pass filters aligned around the carrier frequency (bin 0), as shown in Fig 4. Fig 10 illustrates the process of FFT convolution of a transformed filter impulse response with a transformed input signal. Once the signal is transformed back to the time domain by the inverse FFT, we must then perform a process called the *overlap/add method*. This is because the process of convolution produces an output signal that is equal in length to the sum of the input samples plus the filter taps minus one. I will not attempt to explain the concept here because it is best described in the references.[13]

Fig 11 provides the source code for producing the frequency-domain band-pass filter coefficients. The CalcFilter subroutine is passed the low-frequency cutoff, *FLow*, and the high-frequency cutoff, *FHigh*, for the filter response. The cutoff frequencies are then converted to their respective fractions of the sampling rate for use by the filter-generation routine, nspdFirBandpass. The FFT order is also determined in this subroutine, based on the size of the FFT, NFFT. The nspdFirBandpass computes the impulse response of the band-pass filter of bandwidth *Fl*() to *Fh*() and a length of *FILTERTAPS*. It then places the result in the array variable *Rh*(). The NSP_WinBlackmanOpt causes the impulse response to be windowed by a Blackman window function. For a discussion of windowing, refer to the DSP Guide.[14] The value of one that is passed to the routine causes the result to be normalized.

Next, the impulse response is converted to the frequency domain by nspzrFftNip. The input parameters are *Rh*(), the real part of the impulse response, and *Ih*(), the imaginary part that has been set to zero. NSP_Forw tells the routine to perform the forward FFT. We next convert the frequency-domain result of the FFT, *reH*() and *imH*(), to polar form using the nspdbrCartToPolar routine. The filter magnitudes, *filterM*(), and filter phase, *filterP*(), are stored for use in the FFT fast convolution filter. Other than when we manually change the band-pass filter selection, the filter response does not change. This means that we

```
nspdbMpy2 filterM, M, NFFT        'Multiply Magnitude Bins

nspdbAdd2 filterP, P, NFFT        'Add Phase Bins
```

Figure 12 – FFT Fast Convolution Filtering Code Using Polar Vectors

```
'Compute: RealIn(s) = (RealOut(s) * reH(s)) - (ImagOut(s) * imH(s))
  nspdbMpy3 RealOut, reH, RealOut_1, NFFT
  nspdbMpy3 ImagOut, imH, ImagOut_1, NFFT
  nspdbSub3 RealOut_1, ImagOut_1, RealIn, NFFT         'RealIn for IFFT

  'Compute: ImagIn(s) = (RealOut(s) * imH(s)) + (ImagOut(s) * reH(s))
  nspdbMpy3 RealOut, imH, RealOut_2, NFFT
  nspdbMpy3 ImagOut, reH, ImagOut_2, NFFT
  nspdbAdd3 RealOut_2, ImagOut_2, ImagIn, NFFT         'ImagIn for IFFT
```

Figure 13 – Alternate FFT Fast Convolution Filtering Code Using Cartesian Vectors

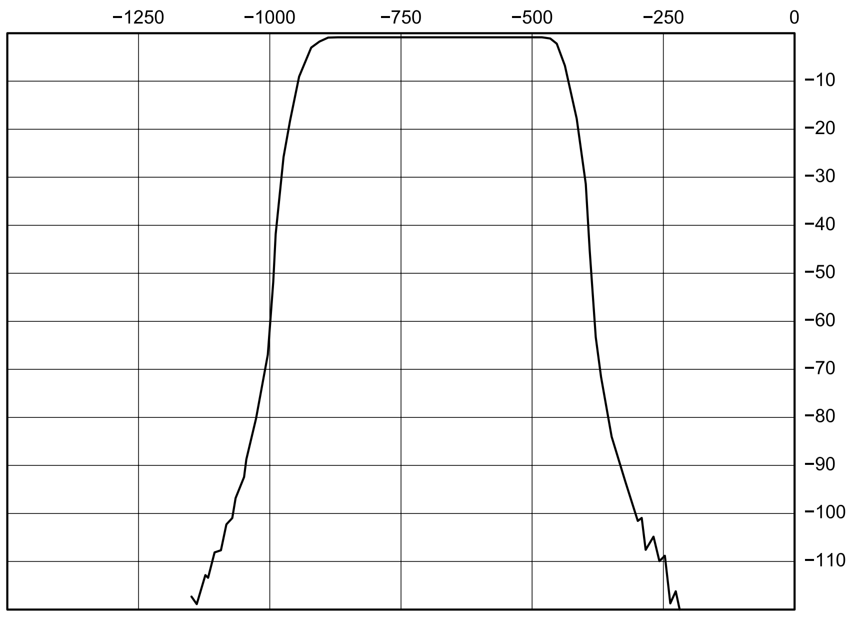

Fig 14—Actual 500-Hz CW filter pass-band display. FFT fast-convolution filtering is used with 2048 filter taps to produce a 1.05 shape factor from 3 dB to 60 dB down and over 120 dB of stop-band attenuation just 250 Hz beyond the 3 dB points.

```
'Convert polar to cartesian
  nspdbrPolarToCart M, P, RealIn, ImagIn, NFFT

  'Inverse FFT to convert back to time domain
  nspzrFftNip RealIn, ImagIn, RealOut, ImagOut, order, NSP_Inv

  'Overlap and Add from last FFT/IFFT:  RealOut(s) = RealOut(s) + Overlap(s)
  nspdbAdd3 RealOut, IOverlap, RealOut, FILTERTAPS - 2
  nspdbAdd3 ImagOut, QOverlap, ImagOut, FILTERTAPS - 2

  'Save Overlap for next pass
  For S = BLKSIZE To NFFT - 1
     IOverlap(S - BLKSIZE) = RealOut(S)
     QOverlap(S - BLKSIZE) = ImagOut(S)
  Next
```

Figure 15 – Inverse FFT and Overlap/Add Code

only have to calculate the filter response once when the filter is first selected by the user.

Fig 12 provides the code for an FFT fast-convolution filter. Using the nspdbMpy2 routine, the signal-spectrum magnitude bins, *M*(), are multiplied by the filter frequency-response magnitude bins, *filterM*(), to generate the resulting in-place filtered magnitude-response bins, *M*(). We then use nspdbAdd2 to add the signal phase bins, *P*(), to the filter phase bins, *filterP*(), with the result stored in-place in the filtered phase-response bins, *P*(). Notice that FFT convolution can also be performed in Cartesian coordinates using the method shown in Fig 13, although this method requires more computational resources. Other uses of the frequency-domain magnitude values include FFT averaging, digital squelch and spectrum display.

Fig 14 shows the actual spectral output of a 500-Hz filter using wide-bandwidth noise input and FFT averaging of the signal over several seconds. This provides a good picture of the frequency response and shape of the filter. The shape factor of the 2048-tap filter is 1.05 from the 3-dB to the 60-dB points (most manufacturers measure from 6 dB to 60 dB, a more lenient specification). Notice that the stop-band attenuation is greater than 120 dB at roughly 250 Hz from the 3-dB points. This is truly a brick-wall filter!

An interesting fact about this method is that the window is applied to the filter impulse response rather than the input signal. The filter response is normalized so signals within the passband are not attenuated in the frequency domain. I believe that this normalization of the filter response removes the usual attenuation associated with windowing the signal before performing the FFT. To overcome such windowing attenuation, it is typical to apply a 50-75% overlap in the time-domain sampling process and average the FFTs in the frequency

```
If AGC = True Then

    ‘If true increment AGCLoop counter, otherwise reset to zero
    AGCLoop = IIf(AGCLoop < AGCHang - 1, AGCLoop + 1, 0)

    nspdbrCartToPolar RealOut, ImagOut, M, P, BLKSIZE ‘Envelope Polar Magnitude

    Vpk = nspdMax(M, BLKSIZE)          ‘Get peak magnitude

    If Vpk <> 0 Then     ‘Check for divide by zero
      G(AGCLoop) = 16384 / Vpk         ‘AGC gain factor with 6 dB headroom
      Gain = nspdMin(G, AGCHang)       ‘Find peak gain reduction (Min)
    End If

    If Gain > MaxGain Then Gain = MaxGain      ‘Limit Gain to MaxGain

    If Gain < PrevGain Then      ‘AGC Gain is decreasing
      GainStep = (PrevGain - Gain) / 44  ’44 Sample ramp = 1 ms attack time
      For S = 0 To 43   ‘Ramp Gain down over 1 ms period
        M(S) = M(S) * (PrevGain - ((S + 1) * GainStep))
      Next
      For S = 44 To BLKSIZE - 1         ‘Multiply remaining Envelope by Gain
        M(S) = M(S) * Gain
      Next
    Else
      If Gain > PrevGain Then   ‘AGC Gain is increasing
        GainStep = (Gain - PrevGain) / 44          ’44 Sample ramp = 1 ms decay time
        For S = 0 To 43            ‘Ramp Gain up over 1 ms period
          M(S) = M(S) * (PrevGain + ((S + 1) * GainStep))
        Next
        For S = 44 To BLKSIZE - 1       ‘Multiply remaining Envelope by Gain
          M(S) = M(S) * Gain
        Next
      Else
        nspdbMpy1 Gain, M, BLKSIZE   ‘Multiply Envelope by AGC gain
      End If
    End If

    PrevGain = Gain     ‘Save Gain for next loop

    nspdbThresh1 M, BLKSIZE, 32760, NSP_GT  ‘Hard limiter to prevent overflow

  End If
```

Figure 16 – Digital AGC Code

domain. I would appreciate comments from knowledgeable readers on this hypothesis.

The IFFT and Overlap/Add—Conversion Back to the Time Domain

Before returning to the time domain, we must first convert back to Cartesian coordinates by using nspdbrPolarToCart as illustrated in Fig 15. Then by setting the NSP_Inv flag, the inverse FFT is performed by nspzrFftNip, which places the time-domain outputs in *RealOut*() and *ImagOut*(), respectively. As discussed previously, we must now overlap and add a portion of the signal from the previous capture cycle as described in the DSP Guide (see Note 13). *Ioverlap*() and *Qoverlap*() store the in-phase and quadrature overlap signals from the last pass to be added to the new signal block using the nspdbAdd3 routine.

Digital AGC with Variable Hang Time

The digital AGC code in Fig 16 provides fast-attack and -decay gain control with variable hang time. Both attack and decay occur in approximately 1 ms, but the hang time may be set to any desired value in increments of 46 ms. I have chosen to implement the attack/decay with a linear ramp function rather than an exponential function as described in DSP communications texts.[15] It works extremely well and is intuitive to code. The flow diagram in Fig 17 outlines the logic used in the AGC algorithm.

Refer to Figs 16 and 17 for the following description. First, we check to see if the AGC is turned on. If so, we increment *AGCLoop*, the counter for AGC hang-time loops. Each pass through the code is equal to a hang time of 46 ms. PC SDR provides hang-time loop settings of 3 (fast, 132 ms), 5 (medium, 230 ms), 7 (slow, 322 ms) and 22 (long, 1.01 s). The hang-time setting is stored in the *AGCHang*variable. Once the hang-time counte resets, the decay

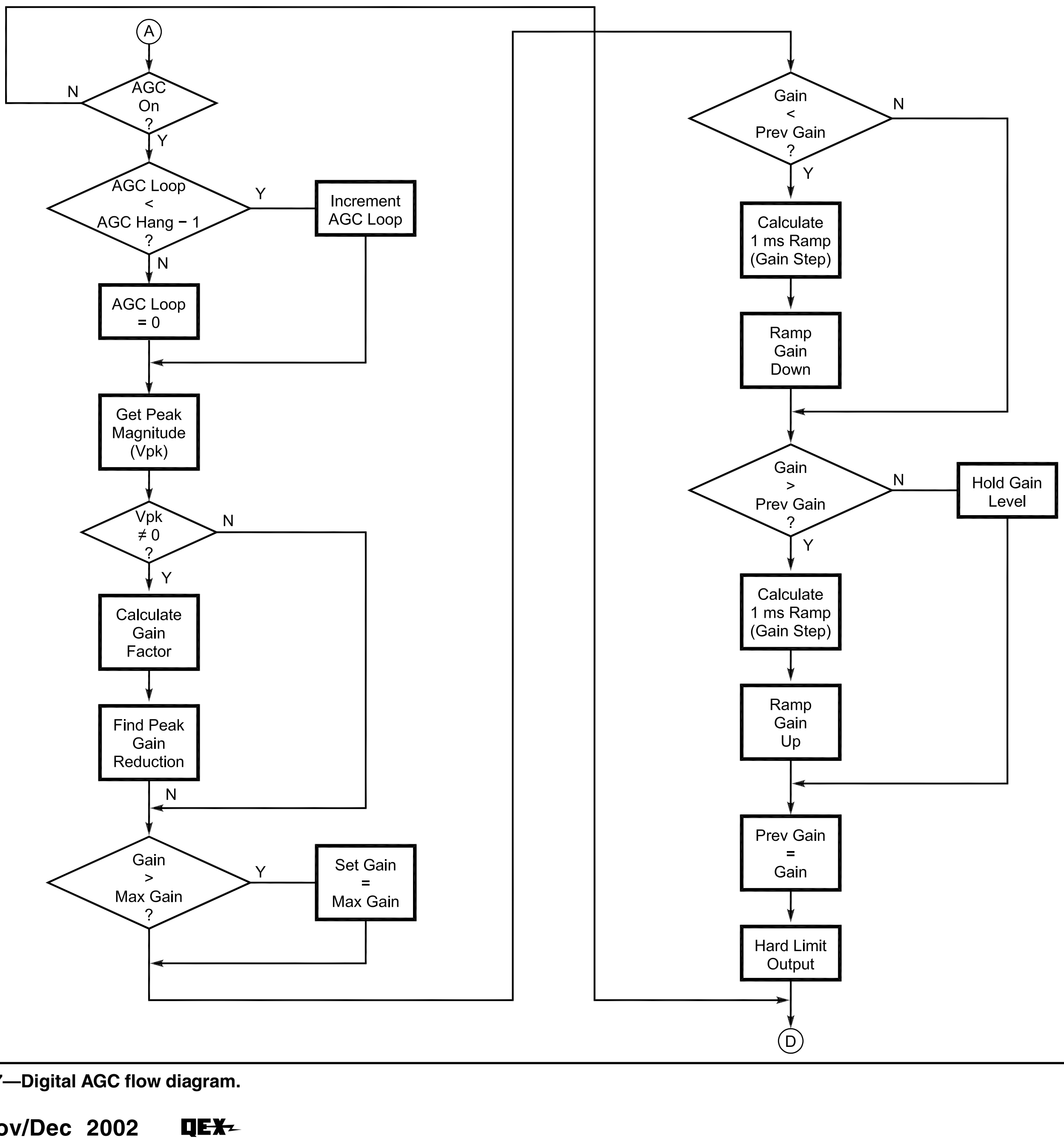

Fig 17—Digital AGC flow diagram.

occurs on a 1-ms linear slope.

To determine the AGC gain requirement, we must detect the envelope of the demodulated signal. This is easily accomplished by converting from Cartesian to polar coordinates. The value of *M*() is the envelope, or magnitude, of the signal. The phase vector can be ignored insofar as AGC is concerned. We will need to save the phase values, though, for conversion back to Cartesian coordinates later. Once we have the magnitudes stored in *M*(), it is a simple matter to find the peak magnitude and store it in *Vpk* with the function nspdMax. After checking to prevent a divide-by-zero error, we compute a gain factor relative to 50% of the full-scale value. This provides 6 dB of headroom from the signal peak to the full-scale output value of the DAC. On each pass, the gain factor is stored in the *G*() array so that we can find the peak gain reduction during the hang-time period using the nspdMin function. The peak gain-reduction factor is then stored in the *Gain* variable. Note that *Gain* is saved as a ratio and not in decibels, so that no log/antilog conversion is needed.

The next step is to limit *Gain* to the *MaxGain* value, which may be set by the user. This system functions much like an IF-gain control allowing *Gain* to vary from negative values up to the *MaxGain* setting. Although not provided in the example code, it is a simple task to create a front panel control in *Visual Basic* to manually set the *MaxGain* value.

Next, we determine if the gain must be increased, decreased or left unchanged. If *Gain* is less than *PrevGain* (that is the *Gain* setting from the signal block stored on the last pass through the code), we ramp the gain down linearly over 44 samples. This yields an attack time of approximately 1 ms at a 44,100-Hz sampling rate. *GainStep* is the slope of the ramp per sample time calculated from the *PrevGain* and *Gain* values. We then incrementally ramp down the first 44 samples by the *GainStep* value. Once ramped to the new *Gain* value, we multiply the remaining samples by the fixed *Gain* value.

If *Gain* is increasing from the *PrevGain* value, the process is simply reversed. If *Gain* has not changed, all samples are multiplied by the current *Gain* setting. After the signal block has been processed, *Gain* is saved in *PrevGain* for the next signal block. Finally, nspdbThresh1 implements a hard limiter at roughly the maximum output level of the DAC, to prevent overflow of the integer-variable output buffers.

Send the Demodulated or Modulated Signal to the Output Buffer

The final step is to format the processed signal for output to the DAC. When receiving, the *RealOut*() signal is copied, sample by sample, into both the left and right channels. For transmiting, *RealOut*() is copied to the right channel and *ImagOut*() is copied to the left channel of the DAC. If binaural receiving is desired, the *I* and *Q* signal can optionally be sent to the right and left channels respectively, just as in the transmit mode.

Controlling the Demonstration Code

The SDR demonstration code (see Note 3) has a few selected buttons for setting AGC hang time, filter selection and sideband selection. The code for these functions is shown in Fig 18. The code is self-explanatory and easy to modify for additional filters, different hang times and other modes of operation. Feel free to experiment.

The Fully Functional SDR-1000 Software

The SDR-1000, my nomenclature

```
Private Sub cmdAGC_Click(Index As Integer)

  MaxGain = 1000          'Maximum digital gain = 60dB

  Select Case Index

    Case 0
      AGC = True
      AGCHang = 3         '3 x 0.04644 sec = 139 ms
    Case 1
      AGC = True
      AGCHang = 7         '7 x 0.04644 sec = 325 ms
    Case 2
      AGC = False         'AGC Off
  End Select

End Sub

Private Sub cmdFilter_Click(Index As Integer)

  Select Case Index

    Case 0
      CalcFilter 300, 3000        '2.7KHz Filter
    Case 1
      CalcFilter 500, 1000        '500Hz Filter
    Case 2
      CalcFilter 700, 800         '100Hz Filter
  End Select

End Sub

Private Sub cmdMode_Click(Index As Integer)

  Select Case Index

    Case 0      'Change mode to USB
      SSB = True
      USB = True
    Case 1      'Change mode to LSB
      SSB = True
      USB = False
  End Select

End Sub
```

Figure 18 – Control Code for the Demo Front Panel

for the PC SDR, contains a significant amount of code not illustrated here. I have chosen to focus this article on the essential DSP code necessary for modulation and demodulation in the frequency domain. As time permits, I hope to write future articles that delve into other interesting aspects of the software design.

Fig 19 shows the completed front-panel display of the SDR-1000. I have had a great deal of fun creating—and modifying many times—this user interface. Most features of the user interface are intuitive. Here are some interesting capabilities of the SDR-1000:

- A real-time spectrum display with one-click frequency tuning using a mouse.
- Dual, independent VFOs with database readout of band-plan allocation. The user can easily access and modify the band-plan database.
- Mouse-wheel tuning with the ability to change the tuning rate with a click of the wheel.
- A multifunction digital- and analog-readout meter for instantaneous and average signal strength, AGC gain, ADC input signal and DAC output signal levels.
- Extensive VFO, band and mode control. The band-switch buttons also provide a multilevel memory on the same band. This means that by pressing a given band button multiple times, it will cycle through the last three frequencies visited on that band.
- Virtually unlimited memory capability is provided through a Microsoft *Access* database interface. The memory includes all key settings of the radio by frequency. Frequencies may also be grouped for scanning.
- Ten standard filter settings are provided on the front panel, plus independent, continuously variable filters for both CW and SSB.
- Local and UTC real-time clock displays.
- Given the capabilities of *Visual Basic*, the possibility for enhancement of the user interface is almost limitless. The hard part is "shooting the engineer" to get him to stop designing and get on the air.

There is much more that can be accomplished in the DSP code to customize the PC SDR for a given application. For example, Leif Åsbrink, SM5BSZ, is doing interesting weak-signal moonbounce work under *Linux*.[16]

Also, Bob Larkin, W7PUA, is using the DSP-10 he first described in the September, October and November 1999 issues of *QST* to experiment with weak-signal, over-the-horizon microwave propagation.[17]

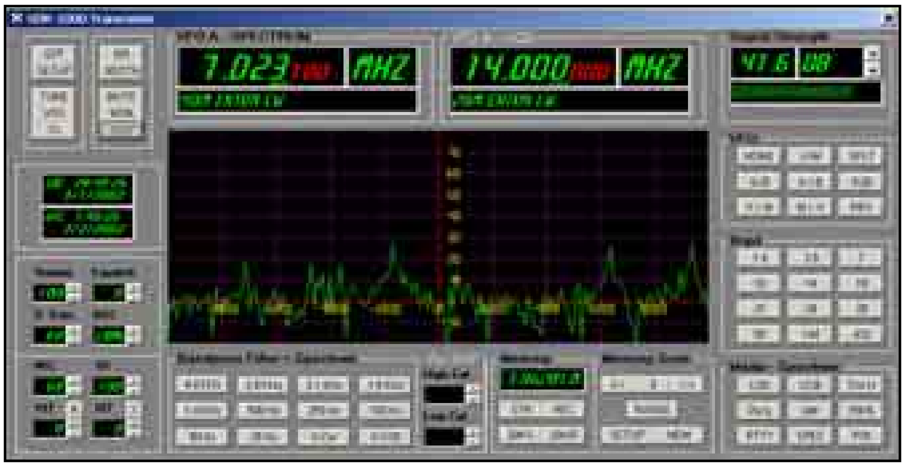

Fig 19—SDR-1000 front-panel display.

Coming in the Final Article

In the final article, I plan to describe ongoing development of the SDR-1000 hardware. Included will be a tradeoff analysis of gain distribution, noise figure and dynamic range. I will also discuss various approaches to analog AGC and explore frequency control using the AD9854 quadrature DDS.

Several readers have indicated interest in PC boards. To date, all prototype work has been done using "perfboards." At least one reader has produced a circuit board, that person is willing to make boards available to other readers. If you e-mail me, I will gladly put you in contact with those who have built boards. I also plan to have a Web site up and running soon to provide ongoing updates on the project.

Notes

[1]G. Youngblood, AC5OG, "A Software Defined Radio for the Masses, Part 1," *QEX*, Jul/Aug 2002, pp 13-21.

[2]G. Youngblood, AC5OG, "A Software Defined Radio for the Masses, Part 2," *QEX*, Sep/Oct 2002, pp 10-18.

[3]The demonstration source code for this project may be downloaded from *ARRLWeb* at **www.arrl.org/qexfiles/**. Look for 1102Youngblood.zip.

[4]The functions of the Intel Signal Processing Library are now provided in the Intel Performance Primitives (Version 3.0, beta) package for Pentium processors and Itanium architectures. The IPP is available free to be downloaded from **developer. intel.com/software/products/ipp/ipp30/index.htm**.

[5]D. Hershberger, W9GR, and Dr S. Reyer, WA9VNJ, "Using The LMS Algorithm For QRM and QRN Reduction," *QEX*, Sep 1992, pp 3-8.

[6]D. Hall, KF4KL, "Spectral Subtraction for Eliminating Noise from Speech," *QEX*, Apr 1996, pp 17-19.

[7]J. Bloom, KE3Z, "Correlation of Sampled Signals," *QEX*, Feb 1996, pp 24-28.

[8]R. Lyons, *Understanding Digital Signal Processing* (Reading, Massachusetts: Addison-Wesley, 1997) pp 133, 330-340, 429-430.

[9]D. Smith, KF6DX, *Digital Signal Processing Technology* (Newington, Connecticut: ARRL, 2001; ISBN: 0-87259-819-5; Order #8195) pp 4-1 through 4-15.

[10]D. Smith, KF6DX, "Signals, Samples and Stuff: A DSP Tutorial (Part 1)," *QEX* (Mar/Apr 1998), pp 5-6.

[11]Information on FFT convolution may be found in the following references: R. Lyons, *Understanding Digital Signal Processing*, (Addison-Wesley, 1997) pp 435-436; M. Frerking, *Digital Signal Processing in Communication Systems* (Boston, Massachusetts: Kluwer Academic Publishers) pp 202-209; and S. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing* (San Diego, California: California Technical Publishing) pp 311-318.

[12]S. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing* (California Technical Publishing) pp 107-122. This is available for free download at **www.DSPGuide.com**.

[13]Overlap/add method: Ibid, Chapter 18, pp 311-318; M. Freirking, pp 202-209.

[14]S. Smith, Chapter 9, pp 174-177.

[15]M. Frerking, *Digital Signal Processing in Communication Systems*, (Kluwer Academic Publishers) pp 237, 292-297, 328, 339-342, 348.

[16]See Leif Åsbrink's, SM5BSZ, Web site at **ham.te.hik.se/homepage/sm5bsz/**.

[17]See Bob Larkin's, W7PUA, homepage at **www.proaxis.com/~boblark/dsp10.htm**.