A Software-Defined Radio for the Masses, Part 2

Come learn how to use a PC sound card to enter the wonderful world of digital signal processing.

By Gerald Youngblood, AC5OG

Part 1 gave a general description of digital signal processing (DSP) in software-defined radios (SDRs).[1] It also provided an overview of a full-featured radio that uses a personal computer to perform all DSP functions. This article begins design implementation with a complete description of software that provides a full-duplex interface to a standard PC sound card.

To perform the magic of digital signal processing, we must be able to convert a signal from analog to digital and back to analog again. Most amateur experimenters already have this capability in their shacks and many have used it for slow-scan television or the new digital modes like PSK31.

Part 1 discussed the power of quadrature signal processing using *in-phase* (*I*) and *quadrature* (*Q*) signals to receive or transmit using virtually any modulation method. Fortunately, all modern PC sound cards offer the perfect method for digitizing the *I* and *Q* signals. Since virtually all cards today provide 16-bit stereo at 44-kHz sampling rates, we have exactly what we need capture and process the signals in software. Fig 1 illustrates a direct quadrature-conversion mixer connection to a PC sound card.

This article discusses complete source code for a DirectX sound-card interface in Microsoft *Visual Basic*. Consequently, the discussion assumes that the reader has some fundamental knowledge of high-level language programming.

[1]Notes appear on page 18.

8900 Marybank Dr
Austin, TX 78750
gerald@sixthmarket.com

Sound Card and PC Capabilities

Very early PC sound cards were low-performance, 8-bit mono versions. Today, virtually all PCs come with 16-bit stereo cards of sufficient quality to be used in a software-defined radio. Such a card will allow us to demodulate, filter and display up to approximately a 44-kHz bandwidth, assuming a 44-kHz sampling rate. (The bandwidth is 44 kHz, rather than 22 kHz, because the use of two channels effectively doubles the sampling rate—*Ed.*) For high-performance applications, it is important to select a card that offers a high dynamic range—on the order of 90 dB. If you are just getting started, most PC sound cards will allow you to begin experimentation, although they

may offer lower performance.

The best 16-bit price-to-performance ratio I have found at the time of this article is the Santa Cruz 6-channel DSP Audio Accelerator from Turtle Beach Inc (**www.tbeach.com**). It offers four 18-bit internal analog-to-digital (A/D) input channels and six 20-bit digital-to-analog (D/A) output channels with sampling rates up to 48 kHz. The manufacturer specifies a 96-dB signal-to-noise ratio (SNR) and better than –91 dB total harmonic distortion plus noise (THD+N). Crosstalk is stated to be –105 dB at 100 Hz. The Santa Cruz card can be purchased from online retailers for under $70.

Each bit on an A/D or D/A converter represents 6 dB of dynamic range, so a 16-bit converter has a theoretical limit of 96 dB. A very good converter with low-noise design is required to achieve this level of performance. Many 16-bit sound cards provide no more than 12-14 effective bits of dynamic range. To help achieve higher performance, the Santa Cruz card uses an 18-bit A/D converter to deliver the 96 dB dynamic range (16-bit) specification.

A SoundBlaster 64 also provides reasonable performance on the order of 76 dB SNR according to PC AV Tech at **www.pcavtech.com**. I have used this card with good results, but I much prefer the Santa Cruz card.

The processing power needed from the PC depends greatly on the signal processing required by the application. Since I am using very-high-performance filters and large fast-Fourier transforms (FFTs), my applications require at least a 400-MHz Pentium II processor with a minimum of 128 MB of RAM. If you require less performance from the software, you can get by with a much slower machine. Since the entry level for new PCs is now 1 GHz, many amateurs have ample processing power available.

Microsoft DirectX versus Windows Multimedia

Digital signal processing using a PC sound card requires that we be able to capture blocks of digitized *I* and *Q* data through the stereo inputs, process those signals and return them to the sound-card outputs in pseudo real time. This is called *full duplex*. Unfortunately, there is no high-level software interface that offers the capabilities we need for the SDR application.

Microsoft now provides two application programming interfaces[2] (APIs) that allow direct access to the sound card under *C++* and *Visual Basic*. The original interface is the Windows Multimedia system using the Waveform Audio API. While my early work was done with the Waveform Audio API, I later abandoned it for the higher performance and simpler interface DirectX offers. The only limitation I have found with DirectX is that it does not currently support sound cards with more than 16-bits of resolution. For 24-bit cards, Windows Multimedia is required. While the Santa Cruz card supports 18-bits internally, it presents only 16-bits to the interface. For information on where to download the DirectX software development kit (SDK) see Note 2.

Circular Buffer Concepts

A typical full-duplex PC sound card allows the simultaneous capture and playback of two or more audio channels (stereo). Unfortunately, there is no high-level code in *Visual Basic* or C++ to directly support full duplex as required in an SDR. We will therefore have to write code to directly control the card through the DirectX API.

DirectX internally manages all low-level buffers and their respective interfaces to the sound-card hardware. Our code will have to manage the high-level DirectX buffers (called DirectSoundBuffer and DirectSoundCaptureBuffer) to provide uninterrupted operation in a multitasking system. The DirectSoundCaptureBuffer stores the digitized signals from the stereo

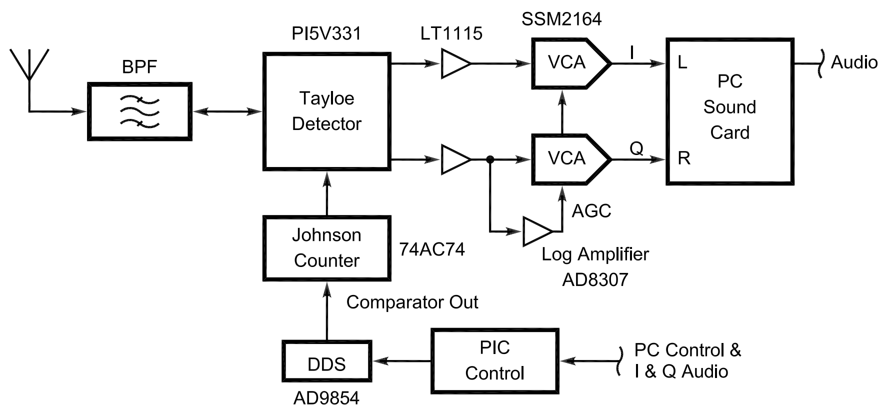

Fig 1—Direct quadrature conversion mixer to sound-card interface used in the author's prototype.

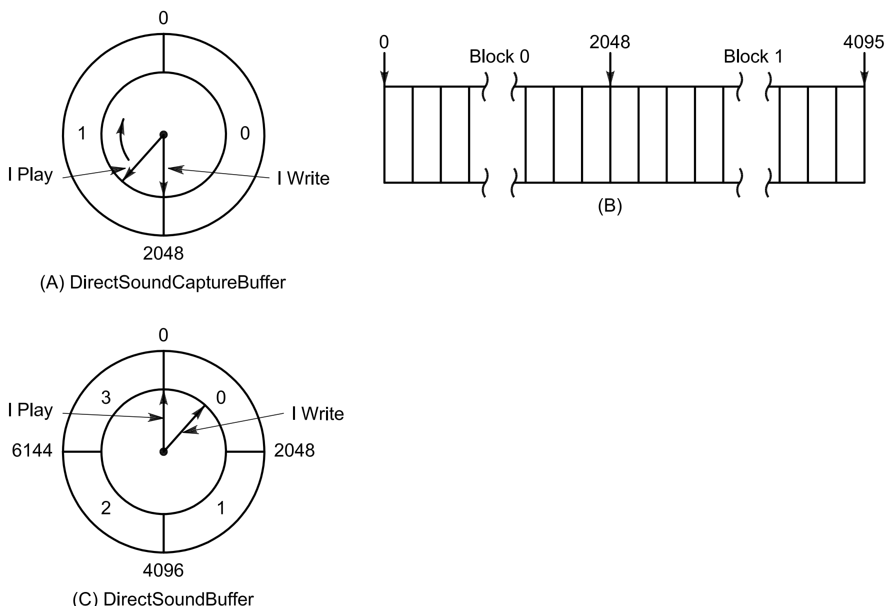

Fig 2—DirectSoundCaptureBuffer and DirectSoundBuffer circular buffer layout.

A/D converter in a circular buffer and notifies the application upon the occurrence of predefined events. Once captured in the buffer, we can read the data, perform the necessary modulation or demodulation functions using DSP and send the data to the DirectSoundBuffer for D/A conversion and output to the speakers or transmitter.

To provide smooth operation in a multitracking system without audio popping or interruption, it will be necessary to provide a multilevel buffer for both capture and playback. You may have heard the term double buffering. We will use double buffering in the DirectSoundCaptureBuffer and quadruple buffering in the DirectSoundBuffer. I found that the quad buffer with overwrite detection was required on the output to prevent overwriting problems when the system is heavily loaded with other applications. Figs 2A and 2B illustrate the concept of a circular double buffer, which is used for the DirectSoundCaptureBuffer. Although the buffer is really a linear array in memory, as shown in Fig 2B, we can visualize it as circular, as illustrated in Fig 2A. This is so because DirectX manages the buffer so that as soon as each cursor reaches the end of the array, the driver resets the cursor to the beginning of the buffer.

The DirectSoundCaptureBuffer is broken into two blocks, each equal in size to the amount of data to be captured and processed between each event. Note that an event is much like an interrupt. In our case, we will use a block size of 2048 samples. Since we are using a stereo (two-channel) board with 16 bits per channel, we will be capturing 8192 bytes per block (2048 samples × 2 channels × 2 bytes). Therefore, the DirectSoundCaptureBuffer will be twice as large (16,384 bytes).

Since the DirectSoundCapture Buffer is divided into two data blocks, we will need to send an event notification to the application after each block has been captured. The DirectX driver maintains cursors that track the position of the capture operation at all times. The driver provides the means of setting specific locations within the buffer that cause an event to trigger, thereby telling the application to retrieve the data. We may then read the correct block directly from the DirectSoundCaptureBuffer segment that has been completed.

Referring again to Fig 2A, the two cursors resemble the hands on a clock face rotating in a clockwise direction. The capture cursor, lPlay, represents the point at which data are currently being captured. (I know that sounds backward, but that is how Microsoft defined it.) The read cursor, lWrite, trails the capture cursor and indicates the point up to which data can safely be read. The data after lWrite and up to and including lPlay are not necessarily good data because of hardware buffering. We can use the lWrite cursor to trigger an event that tells the software to read each respective block of data, as will be discussed later in the article. We will therefore receive two events per revolution of the circular buffer. Data can be captured into one half of the buffer while data are being read from the other half.

Fig 2C illustrates the DirectSoundBuffer, which is used to output data to the D/A converters. In this case, we will use a quadruple buffer to allow plenty of room between the currently playing segment and the segment being written. The play cursor, lPlay, always points to the next byte of data to be played. The write cursor, lWrite, is the point after which it is safe to write data into the buffer. The cursors may be thought of as rotating in a clockwise motion just as the capture cursors do. We must monitor the location of the cursors before writing to buffer locations between the cursors to prevent overwriting data that have already been committed to the hardware for playback.

Now let's consider how the data maps from the DirectSoundCaptureBuffer to the DirectSoundBuffer. To prevent gaps or pops in the sound due to processor loading, we will want to fill the entire quadruple buffer before starting the playback looping. DirectX allows the application to set the starting point for the lPlay cursor and to start the playback at any time. Fig 3 shows how the data blocks map sequentially from the DirectSoundCaptureBuffer to the DirectSoundBuffer. Block 0 from the DirectSoundCaptureBuffer is transferred to Block 0 of the DirectSoundBuffer. Block 1 of the DirectSoundCaptureBuffer is next transferred to Block 1 of the DirectSoundBuffer and so forth. The subsequent source-code examples show how control of the buffers is accomplished.

Full Duplex, Step-by-Step

The following sections provide a detailed discussion of full-duplex DirectX implementation. The example code captures and plays back a stereo audio signal that is delayed by four

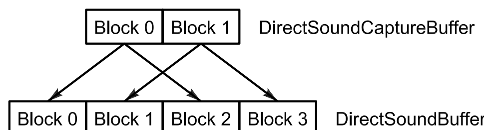

Fig 3—Method for mapping the DirectSoundCaptureBuffer to the DirectSoundBuffer.

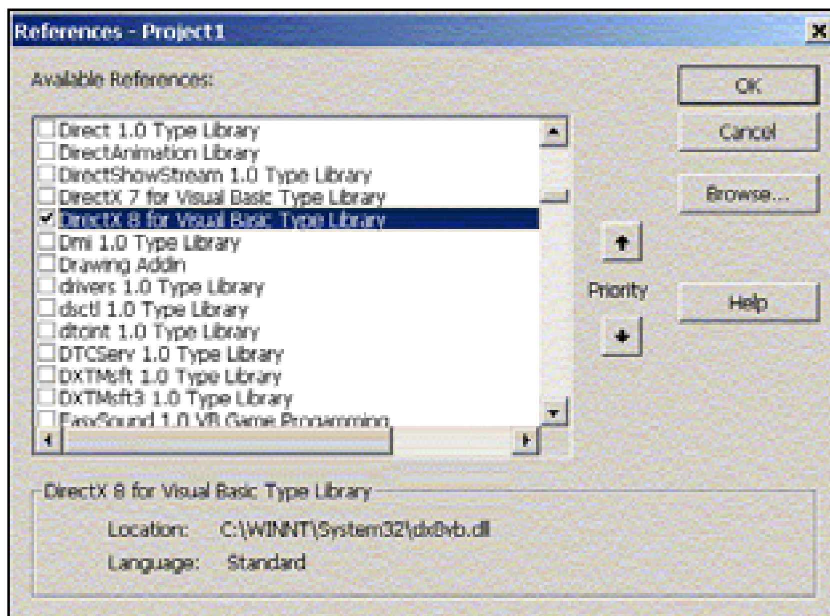

Fig 4—Registration of the DirectX8 for *Visual Basic* Type Library in the *Visual Basic* IDE.

capture periods through buffering. You should refer to the "DirectX Audio" section of the *DirectX 8.0 Programmers Reference* that is installed with the DirectX software developer's kit (SDK) throughout this discussion. The DSP code will be discussed in the next article of this series, which will discuss the modulation and demodulation of quadrature signals in the SDR. Here are the steps involved in creating the DirectX interface:

- Install DirectX runtime and SDK.
- Add a reference to DirectX8 for *Visual Basic* Type Library.
- Define Variables, I/O buffers and DirectX objects.
- Implement DirectX8 events and event handles.
- Create the audio devices.
- Create the DirectX events.
- Start and stop capture and play buffers.
- Process the DirectXEvent8.
- Fill the play buffer before starting playback.
- Detect and correct overwrite errors.
- Parse the stereo buffer into *I* and *Q* signals.
- Destroy objects and events on exit.

Complete functional source code for the DirectX driver written in Microsoft *Visual Basic* is provided for download from the *QEX* Web site.[3]

Install DirectX and Register it within Visual Basic

The first step is to download the DirectX driver and the DirectX SDK

```
Option Explicit

'Define Constants
Const Fs As Long = 44100        'Sampling frequency Hz
Const NFFT As Long = 4096       'Number of FFT bins
Const BLKSIZE As Long = 2048    'Capture/play block size
Const CAPTURESIZE As Long = 4096      'Capture Buffer size

'Define DirectX Objects
Dim dx As New DirectX8          'DirectX object
Dim ds As DirectSound8          'DirectSound object
Dim dspb As DirectSoundPrimaryBuffer8 'Primary buffer object
Dim dsc As DirectSoundCapture8  'Capture object
Dim dsb As DirectSoundSecondaryBuffer8'Output Buffer object
Dim dscb As DirectSoundCaptureBuffer8 'Capture Buffer object

'Define Type Definitions
Dim dscbd As DSCBUFFERDESC      'Capture buffer description
Dim dsbd As DSBUFFERDESC        'DirectSound buffer description
Dim dspbd As WAVEFORMATEX       'Primary buffer description
Dim CapCurs As DSCURSORS        'DirectSound Capture Cursor
Dim PlyCurs As DSCURSORS        'DirectSound Play Cursor

'Create I/O Sound Buffers
Dim inBuffer(CAPTURESIZE) As Integer  'Demodulator Input Buffer
Dim outBuffer(CAPTURESIZE) As Integer 'Demodulator Output Buffer

'Define pointers and counters
Dim Pass As Long                'Number of capture passes
Dim InPtr As Long               'Capture Buffer block pointer
Dim OutPtr As Long              'Output Buffer block pointer
Dim StartAddr As Long           'Buffer block starting address
Dim EndAddr As Long             'Ending buffer block address
Dim CaptureBytes As Long        'Capture bytes to read

'Define loop counter variables for timing the capture event cycle
Dim TimeStart As Double         'Start time for DirectX8Event loop
Dim TimeEnd As Double           'Ending time for DirectX8Event loop
Dim AvgCtr As Long              'Counts number of events to average
Dim AvgTime As Double           'Stores the average event cycle time

'Set up Event variables for the Capture Buffer
Implements DirectXEvent8        'Allows DirectX Events
Dim hEvent(1) As Long           'Handle for DirectX Event
Dim EVNT(1) As DSBPOSITIONNOTIFY      'Notify position array
Dim Receiving As Boolean        'In Receive mode if true
Dim FirstPass As Boolean        'Denotes first pass from Start
```

Fig 5—Declaration of variables, buffers, events and objects. This code is located in the General section of the module or form.

from the Microsoft Web site (see Note 3). Once the driver and SDK are installed, you will need to register the DirectX8 for *Visual Basic* Type Library within the *Visual Basic* development environment.

If you are building the project from scratch, first create a *Visual Basic* project and name it "Sound." When the project loads, go to the Project Menu/ References, which loads the form shown in Fig 4. Scroll through Available References until you locate the DirectX8 for *Visual Basic* Type Library and check the box. When you press "OK," the library is registered.

Define Variables, Buffers and DirectX Objects

Name the form in the Sound project frmSound. In the General section of frmSound, you will need to declare all of the variables, buffers and DirectX objects that will be used in the driver interface. Fig 5 provides the code that is to be copied into the General section. All definitions are commented in the code and should be self-explanatory when viewed in conjunction with the subroutine code.

Create the Audio Devices

We are now ready to create the DirectSound objects and set up the format of the capture and play buffers. Refer to the source code in Fig 6 during the following discussion.

The first step is to create the DirectSound and DirectSoundCapture

```
'Set up the DirectSound Objects and the Capture and Play Buffers
Sub CreateDevices()

    On Local Error Resume Next

    Set ds = dx.DirectSoundCreate(vbNullString)     'DirectSound object
    Set dsc = dx.DirectSoundCaptureCreate(vbNullString)   'DirectSound Capture

    'Check to se if Sound Card is properly installed
    If Err.Number <> 0 Then
        MsgBox "Unable to start DirectSound. Check proper sound card installation"
        End
    End If

    'Set the cooperative level to allow the Primary Buffer format to be set
    ds.SetCooperativeLevel Me.hWnd, DSSCL_PRIORITY

    'Set up format for capture buffer
    With dscbd
        With .fxFormat
            .nFormatTag = WAVE_FORMAT_PCM
            .nChannels = 2                  'Stereo
            .lSamplesPerSec = Fs            'Sampling rate in Hz
            .nBitsPerSample = 16            '16 bit samples
            .nBlockAlign = .nBitsPerSample / 8 * .nChannels
            .lAvgBytesPerSec = .lSamplesPerSec * .nBlockAlign
        End With
        .lFlags = DSCBCAPS_DEFAULT
        .lBufferBytes = (dscbd.fxFormat.nBlockAlign * CAPTURESIZE) 'Buffer Size
        CaptureBytes = .lBufferBytes \ 2    'Bytes for 1/2 of capture buffer
    End With

    Set dscb = dsc.CreateCaptureBuffer(dscbd)      'Create the capture buffer

    ' Set up format for secondary playback buffer
    With dsbd
        .fxFormat = dscbd.fxFormat
        .lBufferBytes = dscbd.lBufferBytes * 2     'Play is 2X Capture Buffer Size
        .lFlags = DSBCAPS_GLOBALFOCUS Or DSBCAPS_GETCURRENTPOSITION2
    End With

    dspbd = dsbd.fxFormat                   'Set Primary Buffer format
    dspb.SetFormat dspbd                    'to same as Secondary Buffer

    Set dsb = ds.CreateSoundBuffer(dsbd)     'Create the secondary buffer

End Sub
```

Fig 6—Create the DirectX capture and playback devices.

objects. We then check for an error to see if we have a compatible sound card installed. If not, an error message would be displayed to the user. Next, we set the cooperative level DSSCL_ PRIORITY to allow the Primary Buffer format to be set to the same as that of the Secondary Buffer. The code that follows sets up the DirectSoundCaptureBufferDescription format and creates the DirectSoundCaptureBuffer object. The format is set to 16-bit stereo at the sampling rate set by the constant *F*s.

Next, the DirectSoundBufferDescription is set to the same format as the DirectSoundCaptureBufferDescription. We then set the Primary Buffer format to that of the Secondary Buffer before creating the DirectSoundBuffer object.

Set the DirectX Events

As discussed earlier, the DirectSoundCaptureBuffer is divided into two blocks so that we can read from one block while capturing to the other. To do so, we must know when

```
'Set events for capture buffer notification at 0 and 1/2
Sub SetEvents()

    hEvent(0) = dx.CreateEvent(Me)          'Event handle for first half of buffer
    hEvent(1) = dx.CreateEvent(Me)          'Event handle for second half of buffer

    'Buffer Event 0 sets Write at 50% of buffer
    EVNT(0).hEventNotify = hEvent(0)
    EVNT(0).lOffset = (dscbd.lBufferBytes \ 2) - 1'Set event to first half of capture buffer

    'Buffer Event 1 Write at 100% of buffer
    EVNT(1).hEventNotify = hEvent(1)
    EVNT(1).lOffset = dscbd.lBufferBytes - 1      'Set Event to second half of capture buffer

    dscb.SetNotificationPositions 2, EVNT()       'Set number of notification positions to 2

End Sub
```

Fig 7—Create the DirectX events.

```
'Create Devices and Set the DirectX8Events
Private Sub Form_Load()
    CreateDevices                      'Create DirectSound devices
    SetEvents                          'Set up DirectX events
End Sub

'Shut everything down and close application
Private Sub Form_Unload(Cancel As Integer)

    If Receiving = True Then
        dsb.Stop                       'Stop Playback
        dscb.Stop                      'Stop Capture
    End If

    Dim i As Integer
    For i = 0 To UBound(hEvent)        'Kill DirectX Events
        DoEvents
        If hEvent(i) Then dx.DestroyEvent hEvent(i)
    Next

    Set dx = Nothing                   'Destroy DirectX objects
    Set ds = Nothing
    Set dsc = Nothing
    Set dsb = Nothing
    Set dscb = Nothing

    Unload Me

End Sub
```

Fig 8—Create and destroy the DirectSound Devices and events.

DirectX has finished writing to a block. This is accomplished using the DirectXEvent8. Fig 7 provides the code necessary to set up the two events that occur when the lWrite cursor has reached 50% and 100% of the DirectSoundCaptureBuffer.

We begin by creating the two event handles hEvent(0) and hEvent(1). The code that follows creates a handle for each of the respective events and sets them to trigger after each half of the DirectSoundCaptureBuffer is filled. Finally, we set the number of notification positions to two and pass the name of the EVNT() event handle array to DirectX.

The CreateDevices and SetEvents subroutines should be called from the Form_Load() subroutine. The Form_Unload subroutine must stop capture and playback and destroy all of the DirectX objects before shutting down. The code for loading and unloading is shown in Fig 8.

Starting and Stopping Capture/Playback

Fig 9 illustrates how to start and stop the DirectSoundCaptureBuffer. The dscb.Start DSCBSTART_LOOPING command starts the DirectSoundCaptureBuffer in a continuous circular loop. When it fills the first half of the buffer, it triggers the DirectX Event8 subroutine so that the data can be read, processed and sent to the DirectSoundBuffer. Note that the DirectSoundBuffer has not yet been started since we will quadruple buffer the output to prevent processor loading from causing gaps in the output. The FirstPass flag tells the event to start filling the DirectSoundBuffer for the first time before starting the buffer looping.

Processing the Direct-XEvent8

Once we have started the DirectSoundCaptureBuffer looping, the completion of each block will cause the DirectX Event8 code in Fig 10 to be executed. As we have noted, the events will occur when 50% and 100% of the buffer has been filled with data. Since the buffer is circular, it will begin again at the 0 location when the buffer is full to start the cycle all over again. Given a sampling rate of 44,100 Hz and 2048 samples per capture block, the block rate is calculated to be 44,100/2048 = 21.53 blocks/s or one block every 46.4 ms. Since the quad buffer is filled before starting playback the total delay from input to output is 4×46.4 ms = 185.6 ms.

The DirectX Event8_DXCallback event passes the eventid as a variable. The case statement at the beginning of the code determines from the eventid, which half of the DirectSoundCaptureBuffer has just been filled. With that information, we can calculate the starting address for reading each block from the DirectSoundCaptureBuffer to the inBuffer() array with the dscb.ReadBuffer command. Next, we simply pass the inBuffer() to the external DSP subroutine, which returns the processed data in the outBuffer() array.

Then we calculate the StartAddr and EndAddr for the next write location in the DirectSoundBuffer. Before writing to the buffer, we first check to make sure that we are not writing between the lWrite and lPlay cursors, which will cause portions of the buffer to be overwritten that have already been committed to the output. This will result in noise and distortion in the audio output. If an error occurs, the FirstPass flag is set to true and the pointers are reset to zero so that we flush the DirectSoundBuffer and start over. This effectively performs an automatic reset when the processor is overloaded, typically because of graphics intensive applications running alongside the SDR application.

If there are no overwrite errors, we write the outBuffer() array that was returned from the DSP routine to the next StartAddr to EndAddr in the DirectSoundBuffer. Important note: In the sample code, the DSP subroutine call is commented out and the inBuffer() array is passed directly to the DirectSoundBuffer for testing of the code. When the FirstPass flag is set to True, we capture and write four data blocks before starting playback looping with the .SetCurrentPosition 0 and .Play DSBPLAY_LOOPING commands.

The subroutine calls to StartTimer and StopTimer allow the average computational time of the event loop to be displayed in the immediate window. This is useful in measuring the efficiency of the DSP subroutine code that is called from the event. In normal operation, these subroutine calls should be commented out.

Parsing the Stereo Buffer into I and Q Signals

One more step that is required to use the captured signal in the DSP subroutine is to separate or parse the left and right channel data into the *I* and *Q* signals, respectively. This can be accomplished using the code in Fig 11. In 16-bit stereo, the left and right channels are interleaved in the inBuffer() and outBuffer(). The code simply copies the alternating 16-bit integer values to the RealIn()), (same as *I*) and ImagIn(), (same as *Q*) buffers respectively. Now we are ready to perform the magic of digital signal processing that we will discuss in the next article of the series.

Testing the Driver

To test the driver, connect an audio generator—or any other audio device, such as a receiver—to the line input of the sound card. Be sure to mute line-in on the mixer control panel so that you will not hear the audio directly through the operating system. You can open the mixer by double clicking on the speaker icon in the lower right corner of your Windows screen. It is also accessible through the Control Panel.

Now run the Sound application and press the On button. You should hear the audio playing through the driver. It will be delayed about 185 ms from the incoming audio because of the quadruple buffering. You can turn the mute control on the line-in mixer on and off to test the delay. It should sound like an echo. If so, you know that everything is operating properly.

Coming Up Next

In the next article, we will discuss in detail the DSP code that provides

```
'Turn Capture/Playback On
Private Sub cmdOn_Click()
    dscb.Start DSCBSTART_LOOPING    'Start Capture Looping
    Receiving = True                'Set flag to receive mode
    FirstPass = True                'This is the first pass after
Start
    OutPtr = 0                      'Starts writing to first buffer
End Sub

'Turn Capture/Playback Off
Private Sub cmdOff_Click()
    Receiving = False               'Reset Receiving flag
    FirstPass = False               'Reset FirstPass flag
    dscb.Stop                       'Stop Capture Loop
    dsb.Stop                        'Stop Playback Loop
End Sub
```

Fig 9—Start and stop the capture/playback buffers.

```
'Process the Capture events, call DSP routines, and output to Secondary Play Buffer
Private Sub DirectXEvent8_DXCallback (ByVal eventid As Long)

    StartTimer                          'Save loop start time

    Select Case eventid                 'Determine which Capture Block is ready
        Case hEvent(0)
            InPtr = 0                   'First half of Capture Buffer
        Case hEvent(1)
            InPtr = 1                   'Second half of Capture Buffer
    End Select

    StartAddr = InPtr * CaptureBytes    'Capture buffer starting address

    'Read from DirectX circular Capture Buffer to inBuffer
    dscb.ReadBuffer StartAddr, CaptureBytes, inBuffer(0), DSCBLOCK_DEFAULT

    'DSP Modulation/Demodulation - NOTE: THIS IS WHERE THE DSP CODE IS CALLED
'    DSP inBuffer, outBuffer

    StartAddr = OutPtr * CaptureBytes   'Play buffer starting address
    EndAddr = OutPtr + CaptureBytes - 1 'Play buffer ending address

    With dsb                            'Reference DirectSoundBuffer

            .GetCurrentPosition PlyCurs 'Get current Play position

            'If true the write is overlapping the lWrite cursor due to processor loading
            If PlyCurs.lWrite >= StartAddr _
                And PlyCurs.lWrite <= EndAddr Then
                    FirstPass = True    'Restart play buffer
                    OutPtr = 0
                    StartAddr = 0
            End If

            'If true the write is overlapping the lPlay cursor due to processor loading
            If PlyCurs.lPlay >= StartAddr _
                And PlyCurs.lPlay <= EndAddr Then
                    FirstPass = True    'Restart play buffer
                    OutPtr = 0
                    StartAddr = 0
            End If

        'Write outBuffer to DirectX circular Secondary Buffer. NOTE: writing inBuffer causes
direct pass through. Replace
        'with outBuffer below to when using DSP subroutine for modulation/demodulation
        .WriteBuffer StartAddr, CaptureBytes, inBuffer(0), DSBLOCK_DEFAULT

        OutPtr = IIf(OutPtr >= 3, 0, OutPtr + 1)      'Counts 0 to 3

        If FirstPass = True Then        'On FirstPass wait 4 counts before starting
            Pass = Pass + 1             'the Secondary Play buffer looping at 0
            If Pass = 3 Then            'This puts the Play buffer three Capture cycles
                FirstPass = False       'after the current one
                Pass = 0                'Reset the Pass counter
                .SetCurrentPosition 0   'Set playback position to zero
                .Play DSBPLAY_LOOPING   'Start playback looping
            End If
        End If

    End With

    StopTimer                           'Display average loop time in immediate window

End Sub
```

Fig 10—Process the DirectXEvent8 event. Note that the example code passes the inBuffer() directly to the DirectSoundBuffer without processing. The DSP subroutine call has been commented out for this illustration so that the audio input to the sound card will be passed directly to the audio output with a 185 ms delay. Destroy objects and events on exit.

```
Erase RealIn, ImagIn

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

Fig 11—Code for parsing the stereo inBuffer() into in-phase and quadrature signals. This code must be imbedded into the DSP subroutine.

modulation and demodulation of SSB signals. Included will be source code for implementing ultra-high-performance variable band-pass filtering in the frequency domain, offset baseband IF processing and digital AGC.

Notes

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

[2]Information on both DirectX and Windows Multimedia programming can be accessed on the Microsoft Developer Network (MSDN) Web site at **www.msdn. microsoft.com/library**. To download the DirectX Software Development Kit go to **msdn.microsoft.com/downloads/** and click on "Graphics and Multimedia" in the left-hand navigation window. Next click on "DirectX" and then "DirectX 8.1" (or a later version if available). The DirectX runtime driver may be downloaded from **www. microsoft.com/windows/directx/downloads/default.asp**.

[3]You can download this package from the ARRL Web **www.arrl.org/qexfiles/**. Look for 0902Youngblood.zip. □□