DSP Programming using DirectSound and MFC/VC++

Frank Perkins, WB5IPM
wb5ipm@comcast.net

Abstract

This paper discusses DSP programming using Microsoft DirectSound and MFC/VC++. Topics covered include an overview of Microsoft Windows programming, the MFC/VC++ framework, DirectSound, AX.25 demodulation and packet decoding, and simple TCP/IP Winsock communications. A packet monitoring program is used as an example. The link to the VC++ folder for this program can be found at www.tapr.org in the "Conferences" section.

Keywords

APRS, AX.25, DirectSound, DSP, MFC, VC++

Introduction

My first experience with DSP programming was writing a set of modems for the TAPR/AMSAT DSP-93 some 10 years ago[1]. Since that time, PC performance has increased to the point that DSP applications can be easily run in real time, and a number of Amateur Radio DSP applications have been published for the PC in recent years. These applications, and a challenge from N5EG, encouraged me to try porting one of my DSP-93 modems to Windows. This project took much longer than I expected, and was much more interesting than I expected. Hopefully what I cover in this paper will help shorten the learning curve on your first PC-based DSP application.

Microsoft Windows Programming

Windows provides your application program with access to hundreds of utility functions, referred to as Application Program Interface functions or API functions. API functions let you do such things as create application windows, draw text and graphics, send and receive TCP/IP packets, and interact with soundcards. When your program is running, Windows will send it messages about events such as mouse clicks and keystrokes. Windows will also allocate slices of CPU time to the one or more "threads of execution" in your program.

To a large extent, "raw" Windows application programming consists of writing code to make API function calls in response to messages sent to your program. In addition to hundreds of API functions, Windows is full of special data structures, variable types and constants. Developing a working knowledge of all these Windows components is a long learning curve for most people. To shorten this learning curve and take much of the tedium out for Windows programming, application programming frameworks, such and Microsoft Visual Basic, Microsoft Foundation Classes in Visual C++ (MFC/VC++) and Borland Delphi have been developed. I have some experience with both Visual Basic and MFC/VC++. I started with Visual Basic, and once event-driven programming started to make sense to me, I found Visual Basic a very intuitive extension of my experience with Microsoft QuickBASIC. However, I decided to switch to MFC/VC++ for my first DSP application based on its reputation for much faster execution.

Coming from a background of QuickBASIC and TMS320C25 (DSP-93) assembly language programming, I found Visual Basic source code examples initially quite confusing. I saw subroutines that seemed totally isolated; nowhere in the source code did I see a main loop or any other code that called them. Eventually I realized these subroutines were, in effect, being called by the Windows operating system itself, in response to events such as clicking on a menu item with the mouse. These subroutines can be thought of as similar to interrupt service routines in assembly language programming. As mentioned above, Windows actually sends event messages to your application, plus several other types of messages. When you are doing "raw" Windows programming (no framework), you use a special "callback" API function which is in your application code but is called by Windows to deliver messages to your application. You then have to decide which messages to act on and how, using a switch statement that can quickly become very large and tedious to implement. Visual Basic, MFC/VC++ and Delphi all simplify this message handling problem.

The MFC/VC++ Framework

Recent versions of Microsoft Visual C++ include the MFC application framework (I am using VC++ 6). MFC is an object-oriented programming framework. Recall that object-oriented programming makes heavy use of classes, which can be thought of as blueprints for specific types of code objects. Classes include variables and other data structures that relate to the same topic, plus the functions (methods) that operate directly on this data. A class can include another class, called a base class, as part of its definition. In this case, the data structures and functions of the base class are inherited by the new (derived) class. The functions inherited from the base class provide the derived class with default functionality, which can be selectively overridden as needed in the derived class. An example of an MFC class is the CString class, which provides string handling capabilities similar to Visual Basic. You can instantiate, or create, a CString object just like an ordinary variable:

```
CString  str;
```

Storage for the characters in the str object is automatically created, and you can apply a whole range of methods to str, such as retrieving the three left characters:

```
CString  lftstr =  str.Left(3);
```

MFC contains dozens of classes that encapsulate much more functionality than single API function calls, and they are generally much simpler to use than API functions. Nothing in MFC prevents you from using API functions, and in some cases you still have to use API function calls.

MFC/VC++ programming is a big topic, so expect to take several months getting familiar with it. The books I have found most helpful on MFC/VC++ programming are listed at the end of the paper[2-12]. The MFC/VC++ source code published by AE4JY[13] is also an excellent resource.

DirectSound

There are two ways MFC/VC++ can interact with the soundcard in your PC; by using the Microsoft multimedia wave API functions or the DirectSound API functions in Microsoft DirectX. The wave API functions were available first. Low level wave API functions are complex to use. Fortunately, AE4JY has developed a class, CSound, that greatly simplifies using the wave API functions in MFC/VC++. I did my first PC-based DSP experiments using his CSound class. DirectSound is a more recent development, and is part of DirectX, which Microsoft has developed to speed up graphic and sound intensive applications such as games. DirectSound is less complex to use than wave sound. AC5OG has demonstrated using DirectSound with Visual Basic in his QEX articles[14], and using DirectSound with MFC/VC++ is discussed in this article.

You have to manually add the libraries for multimedia wave and DirectSound to your MFC/VC++ linker. To do this, start VC++, load your MFC application, click on the "Projects" menu, then click on "Settings" and select the "Link" tab. In the "Object/library modules" text box add:

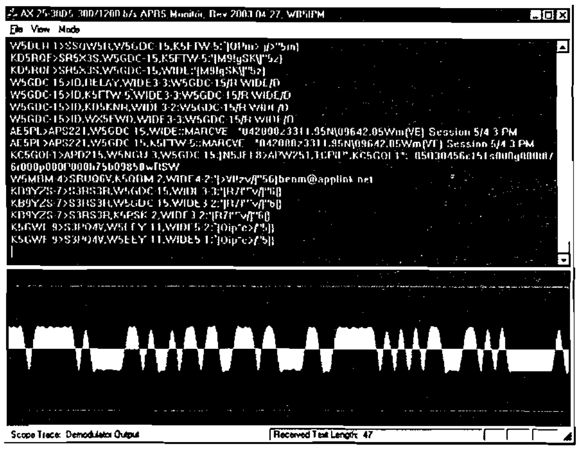
```
winmm.lib dsound.lib dxguid.lib dxerr8.lib
```

Do this for both the "Win32 Debug" and "Win 32 Release" settings. The multimedia library is included with recent versions of MFC/VC++. You can get the latest DirectX software development kit (SDK) from the Microsoft web site. I used DirectX 8 in the example discussed here.

Packet Monitoring Program Operation

Before discussing the design of the packet monitoring program, I will briefly discuss its operation. When the monitor starts, a dialog box is presented to let you choose either to activate or cancel a simple TCP/IP loopback server. I use this server to drive WinAPRS[15]. To make WinAPRS aware of the monitor, put the wb5ipm.prt file in the WinAPRS "Ports" folder. Then to use the monitor with WinAPRS, click on the "Activate TCP/IP Server" button in the monitor's dialog box and start WinAPRS. Click on the "Ports list" menu item of the WinAPRS "Settings" menu, and highlight the "Soundcard Packet Monitor" row. Then click the Open button. You should see ACTIVE appear in the status column of the highlighted row. The monitor will then start feeding packets to WinAPRS. If you simply want to monitor packet text and not run WinAPRS, click on the "Cancel TCP/IP Server" button in to monitor's dialog box.

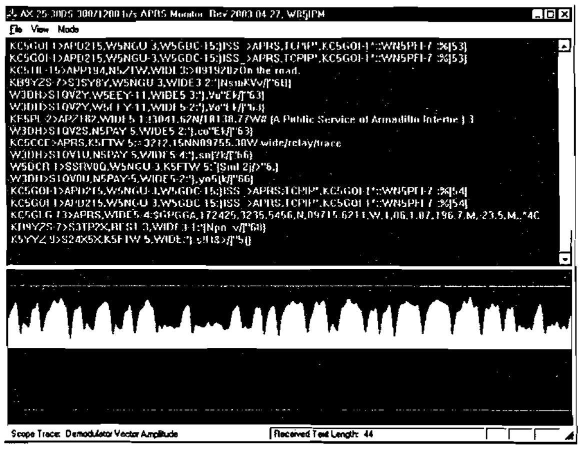
The monitor has three menus, "File", "View" and "Mode". The "File" menu contains "New", "Open", "Save", "Save As", a list of files, and "Exit". Basically normal Windows file support for the screen text. The "View" menu contains the following items: "Clear Text", "Demodulator Output" which is the default scope display showing the output of the FSK demodulator (Figure 1), "Demodulator Vector Amplitude" which shows the vector amplitude of the FSK signal into the demodulator (Figure 2), and "Input FFT and Waveform", which shows the FFT of the input signal on the left and the time waveform on the right (Figure 3). The "Mode" menu allows you to choose either "300 b/s AX.25" or "1200 b/s AX.25" (default). Under the "Mode" menu you can


AX 25-30DS 300/1200 b/s APRS Monitor, Rev 2003 04 27, WB5IPM
File View Mode

```
W5DCR 1>SSQV5R,W5GDC 15,K5FTW 5:'[0Pm> j/>"5m}
KD5ROF>SR5X3S,W5GDC-15,K5FTW-5:'[M9!gSK\]"5z}
KD5ROF>SR5X3S,W5GDC-15,WIDE:'[M9lqSK\]"5z}
W5GDC 15>ID,RELAY,WIDE3-3:W5GDC-15/R WIDE/D
W5GDC-15>ID,K5FTW-5,WIDE3-3:W5GDC-15/R WIDE/D
W5GDC-15>ID,KD5KNR,WIDE 3-2:W5GDC-15/R WIDE/D
W5GDC-15>ID,WX5FWD,WIDE3-3:W5GDC 15/R WIDE/D
AE5PL>APS221,W5GDC 15,WIDE::MARCVE   *042000z3311.95N\09642.05Wm{VE} Session 5/4 3 PM
AE5PL>APS221,W5GDC 15,K5FTW-5::MARCVE   *042000z3311.95N\09642.05Wm{VT} Session 5/4 3 PM
KC5GOI-1>APD215,W5NGU 3,W5GDC 15:}N5JFE 8>APW251,TCPIP*,KC5GOI 1*:  05030456c151s000g00007
6r000p000P000h75b09850wRSW
W5MBM 4>SRUQ6V,K5ODM 2,WIDE 4-2:'[>VPzv/]"5G}benm@applink net
KD9YZS-7>S3RS3R,W5GDC 15,WIDE 3-3:'[R7f"v/]"6}
KB9YZS-7>S3RS3R,W5GDC 15,WIDE3 2:'[R7f"v/]"6}
KB9YZS-7>S3RS3R,K5PSK 2,WIDE 3 2:'[R7f"v/]"6}
K5GWE 9>S3PO4V,W5EEY 11,WIDE5-2:'[Oip"c>/"5|}
K5GWE 9>S3PO4V,W5EEY 11,WIDE5 1:'[Oip"c>/"5|}
|
```

```
Scope Trace: Demodulator Output          Received Text Length: 47
```

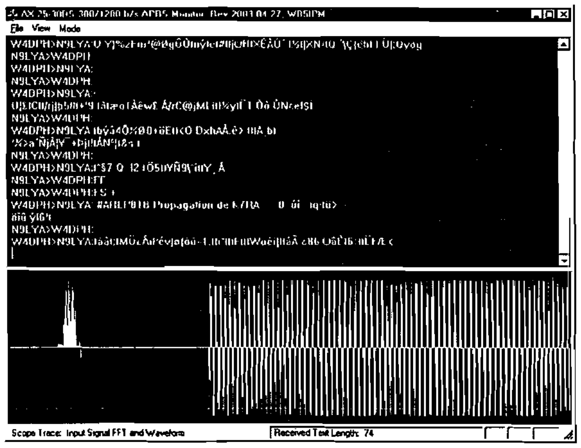
Figure 1



```
KC5GOI 1>APD215,W5NGU 3,W5GDC-15:}ISS  >APRS,TCPIP*,KC5GOI-1*::WN5PFI-7 :%[53}
KC5GOI-1>APD215,W5NGU-3,W5GDC-15:}ISS_>APRS,TCPIP*,KC5GOI-1*::WN5PFI-7 :%[53}
KC5TIL-15>APP194,N5ZTW,WIDE 3:>091920zOn the road.
KB9YZS-7>S3SY8Y,W5NGU 3,WIDE3 2:'[NsmKVv/]"6B}
W3DH>S1QV2Y,W5EEY-11,WIDE5 3:'},Vu"Ek/]"63}
W3DH>S1QV2Y,W5EEY-11,WIDE 5-2:'},Vo"Ek/]"63}
KF5PL-2>AP2182,WIDE5 1:!3041.62N/10138.77W# {A Public Service of Armadillo Interne } 3
W3DH>S1QV2S,N5PAY 5,WIDE5 2:'},co"Ek/]"63}
KC5CCE>APRS,K5FTW 5:=3212.15NN09755.30W wide/relay/trace
W3DH>S1QV1U,N5PAY 5,WIDE5-4:'},sn[?k/]"66}
W5DCR 1>SSRV0Q,W5NGU 3,K5FTW 5:'[Sml 2j/>"6,}
W3DH>S1QV0U,N5PAY-5,WIDE5-2:'},yo5[k/]"66}
KC5GOI-1>APD215,W5NGU-3,W5GDC-15:}ISS  >APRS,TCPIP*,KC5GOI-1*::WN5PFI-7 :%[54}
KC5GOI 1>APD215,W5NGU 3,W5GDC 15:}ISS  >APRS,TCPIP*,KC5GOI 1*::WN5PFI 7 :%[54}
KC5GLG 13>APRS,WIDE5-4:$GPGGA,172425,3235.5456,N,09715.6211,W,1,06,1.07,196.7,M,-23.5,M,,*4C
KB9YZS-7>S3TP2X,REST 3,WIDE 3-1:'[Npn v/]"68}
K5YYZ 9>S24X5X,K5FTW 5,WIDE:'},s!l8>/]"5{}
```

```
Scope Trace: Demodulator Vector Amplitude          Received Text Length: 44
```

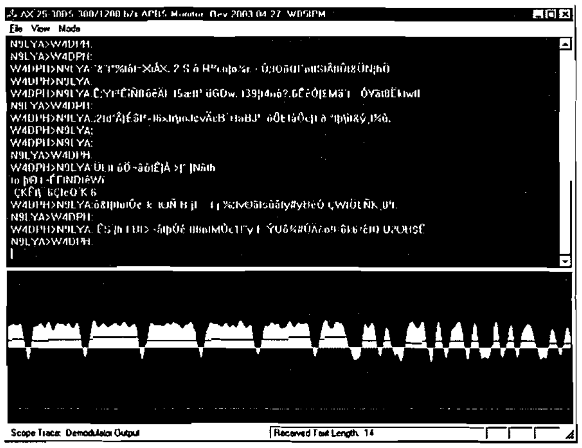
Figure 2



```
W4DPH>N9LYA:U Y}%zFm'@ØgÛÙmýle#ÏjOHDXÉÀÚ` I%|XN4Q `\{ebI I Ú|;Qvóg
N9LYA>W4DPH
W4DPH>N9LYA:
N9LYA>W4DPH:
W4DPH>N9LYA:
U|ZICU/rj|þ5Ø++9 IãtæoIÂêwf Å/rC@)MI ill%yIÌ I Òô ÙN<elSI
N9LYA>W4DPH:
W4DPH>N9LYA:lbýä4Ô%Ø0+öEI<O DxhAÅ.é> IIIA,þI
'%>a`ÑjÅ|Y¯+Þ|IIÂN¶|I8¬ ı
N9LYA>W4DPH:
W4DPH>N9LYA:r^§7 Q  I2 ıÖ5ıYÑ9\'iilY¸Å
N9LYA>W4DPH:FT
N9LYA>W4DPH:ES +
W4DPH>N9LYA: #ARLP018 Propagation de K7RA      0  öi   iq-tü>
ðïü ÿIG'ı
N9LYA>W4DPH:
W4DPH>N9LYA:Jààt:IMÜzÅiPévJø[öü-I.Ilr'IIIFIIWøé|IIãÃ ¿Bb UâÈ'I6:IIÈFÆ <
|
```

```
Scope Trace: Input Signal FFT and Waveform          Received Text Length: 74
```

Figure 3



```
N9LYA>W4DPH.
N9LYA>W4DPH:
W4DPH>N9LYA:`&'I"%IóI-XiÅX. 2 S ó H"ca|o%c · Û;IOôÜI`eüSIÂIIÔI8ÜN;hÖ
W4DPH>N9LYA.
W4DPH>N9LYA:Ĉ;YI"ĈÏÑBóšÅI I5æII' üGDw. I39|I4nó?.ĈĖšÔ|ΣMäT   ÓYäIØÈkIwII
N9LYA>W4DPH:
W4DPH>N9LYA:2Id^Å|ÉâI'-I6xIrjroJevÂcB' IIaBJ'  öÒEIáÔcjI à 'Ilþ|I8ý,I%û.
W4DPH>N9LYA:
W4DPH>N9LYA:
N9LYA>W4DPH.
W4DPH>N9LYA:ÜL II öÖ~âóIÈ|Å >|` |Nãth
Io þ®I-ÉTINDIéWi
 ÇKFI\` bÇIeO'K 6
W4DPH>N9LYA:ö&I|IIuIÜc k IUÑ B |I   I | %:IvØãIsûäIy#yBéÖ ÇWIÕLÑK,ÜI.
N9LYA>W4DPH:
W4DPH>N9LYA: ÊS'|h IIII> -ãIþÜé IIBeIMÜc1I'y I  ÝUô%#ÚA¿oII-ôI6-éIO U?OIISÊ
N9LYA>W4DPH.
|
```

```
Scope Trace: Demodulator Output          Received Text Length: 14
```

Figure 4

also choose "Audio Loopthrough" which allows you to listen to the input audio, albeit somewhat delayed.

Tuning 300 b/s AX.25 on HF (SSB) can be a bit tricky. I start by choosing the "Input FFT and Waveform" display mode and centering the signal between the tick marks on the FFT display. I then switch to the "Demodulator Output" display mode and fine tune my HF rig so that the red "adaptive threshold" line sits on top of the middle green line while a packet is being received (the red line tends to drift up between packets). The adaptive threshold is handy for HF packet monitoring, as there is often some difference in frequency between stations in an HF packet QSO. Figure 4 shows the adaptive threshold operating on a packet that has been deliberately mistuned.

The "Demodulator Vector Amplitude" gives an interesting insight into the amplitude balance between the two FSK tones. I have seen many transmissions that have a 2:1, 3:1, or even higher tone unbalance.

To run the monitor, you must have the end-user runtime files or the SDK for DirectX8 (or higher) installed on your PC. Windows XP includes the DirectX runtime files. For Windows 98 you can download the latest runtime files or SDK from the Microsoft web site. The monitor may not run under Windows 95, even with the latest DirectX runtime files for Windows 95 installed.

Overall Program Design

The packet monitoring program is based on MFC's Single Document Interface (SDI) architecture, which is generated using the AppWizard in VC++. I chose the SDI architecture because it uses a true "industrial strength" overlapped Windows main frame, and it also makes it easy to save received packets as text files. In building the SDI with AppWizard, a Status Bar is added to the bottom of the Main Frame, and CEditView is chosen as the base class for CDSPView to provide text display support in the client area. Then, in the header of the Main Frame class, a CSplitterWnd utility class is added to allow the client area to be split into two rows. The split is defined by overriding the OnCreateClient virtual method in the Main Frame implementation. CEditView is put in the upper split, and a new class, CScope, is added to the program and put in the lower split for waveform display. CScope is derived from MFC's CView class, which provides it graphics support.

To do the DSP, a class called CDSPThread is added to the program, derived from MFC's CWnd base class. CDSPThread is a "worker" thread, and gets its own independent slice of CPU time from the operating system. Soundcard support for CDSPThread is provided by defining another class, CDXSound, which is also derived from MFC's CWnd base class. CDXSound handles calls to the DirectSound API functions. A simple Dialog class, CSockDlg, is also added to allow the user to activate or cancel the TCP/IP loopback server for WinAPRS.

MFC programs that include a worker thread and several views will usually require communication between some of the main classes. Here is a summary of what is done in the example packet monitoring program. CDSPView controls CDSPThread and also handles most of the interfaces with the user, including the TCP/IP server dialog box. The Status Bar belongs to the Main Frame. So to write text to the Status Bar panes, CDSPView uses m_pFrame, which is a pointer to the Main Frame. To create, start, stop and delete CDSPThread, CDSPView uses a global pointer pDSPThread, which is defined above CDSPView's constructor. When CDSPView starts CDSPThread, it gives the thread its pointer so the thread can communicate back to it. CDSPView also contains several data structures which are loaded or read by other classes. CDSPView's m_PktStr is loaded with new packets by CDSPThread, after testing CDSPView's m_fReady flag to be sure CDSPView is not in the middle of processing m_PktStr (poor man's critical section). CDSPThread then posts a user-defined message which triggers CDSPView to process m_PktStr for display. Note that a worker thread can load or read data structures belonging to other classes with a bit of coordination, but it is usually not "safe" for the thread to directly call functions in other classes. Hence the use of messages.

CDSPView's m_fHF flag, which indicates that the monitor is in the HF (300 b/s) mode, is read by CDSPThread and CScope. CDSPView's m_fSO flag, which indicates the user wants the input audio looped through to the audio output, is read by CDSPThread. CDSPView controls the three scope display modes with m_scope, which is read by CScope and CDSPThread. CDSPView reads CSockDlg's m_fIP selection flag and then sets m_fSocket to control the TCP/IP server function, which in turn is read by CDSPThread.

CScope includes two data buffers, m_pData which holds waveform data for display, and and m_pThld, which holds the adaptive threshold data for display. These are loaded by CDSPThread, using the m_pScope pointer, after testing the m_fScope flag to be sure CScope is ready for more data. CScope uses the global pDSPThread pointer to access and update CDSPThread's m_fScope flag. CScope also maintains a pointer to CDSPView, called m_pView, to read CDSPView data as discussed in the above paragraph. CScope also has a Main Frame pointer, m_pFrame, but this is just used as a means to initialize m_pView. CDSPDoc has access to the CEditView text window as a standard feature of the SDI architecture, allowing packets to be stored as text files.

Here are a couple of hints about SDI programs and VC++. Each time you click "New" in the "File" menu, it triggers the OnInitialUpdate method in the first view class (CDSPView in the case of the packet monitoring program). Design the code you add to this method so you do not unintentionally create, define, or initialize things more than one time. I had some very interesting debugging experiences until I figured this out. I use the CEditView control in CDSPView to display text. This control will overflow at about 32,000 characters, and seems to get sluggish on slow computers when it is working with more than about 16,000 characters. So be sure to manage the amount of characters in the CEditView control to avoid a program hang up. The CEditView control only executes a carriage return-line feed when it sees the (char)13-char(10) pair; if just one or the other of these characters occurs, CEditView prints a vertical mark and keeps going. And unless a backspace comes directly from the keyboard, CEditView just prints a vertical mark and goes on. You will have to preprocess any (char)8 backspaces in your own code before adding the text to the CEditView control. In trade for execution speed, VC++ will let you hurt yourself. Become familiar with the terms "memory leak" and "GDI resource leak" and test your code for them. Otherwise, you can have an application appear to run fine for hours and/or for a number of restarts, and then shut your computer down.

DSP Design Details

There are a number of ways to demodulate an FSK signal with DSP. I chose the arctangent method for the packet monitor because of its relative insensitivity to amplitude fluctuations[16]. Figure 5 shows a block diagram of the demodulator. The demodulator and the routines that assemble packets from the demodulated bits are found in CDSPThread::DSPLoop(). Look for the "while (m_fDSPRun == TRUE)" statement within DSPLoop(). The first call in this while loop, "DXS.ReadDirectSound()", feeds the DSP calculations and the packet assembly routines by returning blocks of 4096 audio input samples. Each sample is 16-bit monaural, so the block size is 8192 bytes. After retrieving the block of input samples, either the samples are written back to DirectSound for soundcard output or "silence" is written back, depending on what the user has chosen. The "for (i = 0; i < (int)(DXS.m_BufSize / 2); i++)" loop turns the DSP and packet assembly crank on the 4096 input samples.

Referring again to Figure 5, the audio samples first flow through an IIR high-pass filter to remove any DC offset. Next the samples flow through a limiter, which in retrospect is probably

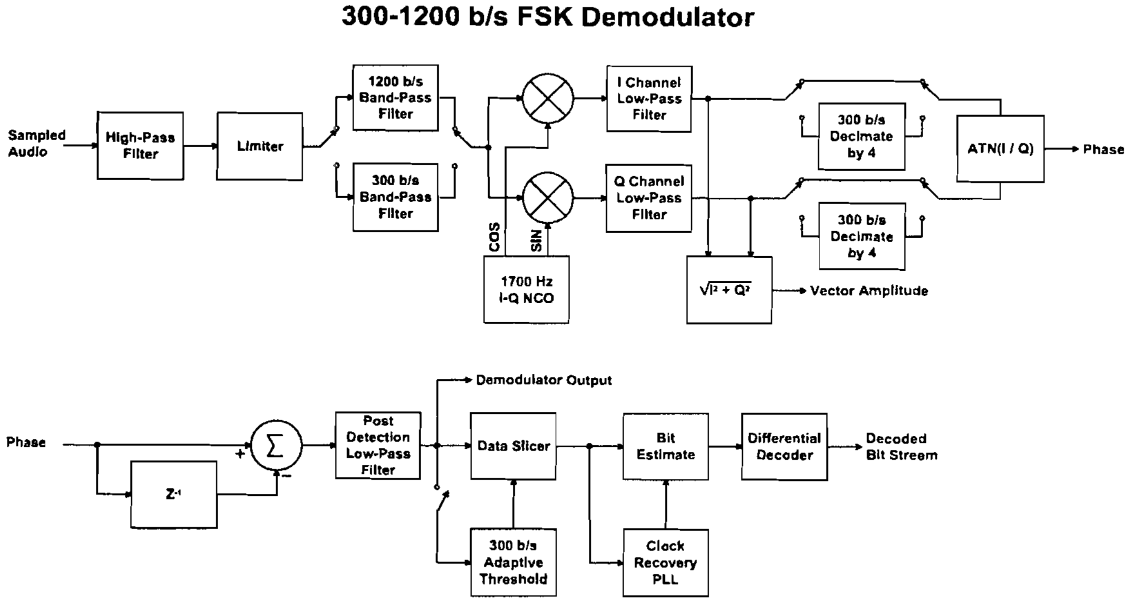


Figure 5

not needed. The samples then flow through an FIR band-pass filter. Note that there are two band-pass filter choices, one for 1200 b/s packet and one for 300 b/s packet. The coefficients for the FIR filters are stored in the DSPData.h header file. Both filters are centered at 1700 Hz. The output of the band-pass filter is applied to two mixers. The local oscillator for one mixer is a 1700 Hz cosine wave (I-channel), and the local oscillator for the other mixer is a 1700 Hz sine wave (Q-channel). The output samples from each mixer are filtered by FIR low-pass filters to remove the "sum" products. The vector amplitude of the signal is calculated at this point by taking the square root of the sum of the squares of the filtered I and Q signals. The vector amplitude is used as a "squelch" for some of the scope displays and is an interesting signal to view itself.

For 1200 b/s packet, the filtered I and Q samples are applied directly to an arctangent phase detector. For 300 b/s packet, the I and Q channels are first decimated by four (300 b/s = ¼ of 1200 b/s). The narrow band-pass filter used for 300 b/s packet prevents aliasing due to this decimation. The prior output of the phase detector is subtracted from the current output of the phase detector to detect the frequency (change in phase between samples) of the audio signal with respect to 1700 Hz. The output samples from the frequency detector flow through a "post detection" low-pass FIR filter. The output of the post-detection filter is the default scope display. The output of the post-detection filter is applied to a data slicer, which is a one-bit digitizer. An input sample equal to or greater than the threshold value of the slicer is digitized at a "1" value, otherwise it is digitized as a "0" value. In the case of 1200 b/s packet, the slicer threshold is set to zero, which represents 1700 Hz, or the frequency half-way between the 1200 and 2200 Hz FSK tones used for 1200 b/s packet. 1200 b/s packet is normally transmitted FM, so the tones you send are the tones you get.

In the case of 300 b/s HF packet, SSB reception is used and the FSK tones received depend on the transmitter frequency and the receiver tuning. A data slicer provides the best noise rejection when its threshold is set half-way between the demodulator output values for each tone. As discussed earlier, an adaptive threshold is used for 300 b/s packet monitoring, to track out frequency offsets between the stations in a QSO. The adaptive threshold used is from N5EG's book on Wireless Digital Communications[17].

Each bit value is "estimated" to be equal to the value of a slicer output sample near the middle of its bit period. To do this estimation, a simple phase-locked loop (PLL) is used to line up the period of a numerically-controlled oscillator (NCO) with the average position of the edges (sample-to-sample transitions from 1 to 0 or 0 to 1) in the slicer output stream. Operation of the PLL is similar to a "bang-bang" feedback controller such as thermostat. When an edge occurs in the slicer output stream, the NCO is tested to see if it is in the first half of its period or in the last half. If the edge occurred in the first half, the NCO is retarded about 6%; if it is in the second half, it is advanced about 6%. This has the effect of sliding the NCO gradually either backwards or forwards into alignment with the embedded clock in the slicer output stream. Once aligned, the PLL will dither back and fourth slightly around the alignment point. Noise has the effect of jittering the edge positions in the slicer output samples, but the average edge position can still be found using filtering effect of the PLL. The PLL will also track out mild errors in the embedded clock rate of the slicer output samples. At the beginning of each NCO period, counter "n" is reset and is then incremented on each sample. The "n" counter reaches 5 near the middle of the bit period (PLL locked), and the value of corresponding sample is used as the bit estimate. Another way to estimate the bit value is to use the "integrate and dump" method, which can be implemented by adding just a few more lines of code. I will leave this as your homework assignment.

AX.25 data is usually differentially encoded, which means a "0" bit is encoded as a change in bit value and a "1" bit is encoded as no change in bit value. Differential decoding just reverses this process. The output of the differential decoder is a stream of bits that you can use to assemble received packets for display and to drive WinAPRS.

Before looking at packet assembly, let's discuss a few points about the CDXSound class that supports the DSP calculations. Recall that "DXS.ReadDirectSound()" is the first call at the top of the "while (m_fDSPRun == TRUE)" loop. If you look at the ReadDirectSound() method in CDXSound, you will see the ::WaitForMultipleObjects function call. This function is looking for two events that are set up when DirectSound is initialized. One event occurs when the lower block of the DirectSound capture buffer (circular) is full of new audio samples, and the other event occurs when the upper block of the DirectSound capture buffer is full of new audio samples. When either of these events occur, the new audio samples are copied into the m_pInBuf buffer for use by the DSP calculations, etc. The ::WaitForMultipleObjects function has a couple of noteworthy features. First, Windows will devote very little CPU time to this function until one of the event objects becomes true. Second, you can make this function time out so you can back out of your application if some other application grabs the soundcard from you or otherwise hogs most of the CPU time for an extended period. Third, this function provides the basic pacing for receiving audio samples, sending audio samples, and doing the DSP calculations and packet assembly routines.

The DirectSound output play buffer is set at twice the size of the input capture buffer, and is subdivided into four blocks. When WriteDirectSound() is called, output samples are written two blocks ahead (circular) of the block where the play cursor is currently located. This approach introduces some delay in the audio output, but it is simple to code and recovers well from CPU loading peaks that cause the audio to get "out of sync". Someday I will rework this call for less delay when I start writing transmit routines.

CDXSound is hard coded for 16-bit monaural sampling at 11,025 samples/second. The block size can be set when an object of CDXSound is instantiated. I have been using a block size of 4,096 samples, or 8,192 bytes. This is 371.5 milliseconds of audio. Using this block size, the packet monitoring program will even run on my old 166 MHz desktop PC. A smaller block size can be used if you are only going to run on fast PCs.

Going back to the topic of packet assembly, the decoded bit stream is first fed into a flag detection correlator. When a flag is detected, the number of accumulated bytes is checked. If there are more than 16 bytes and the FCS tests OK, the packet is assembled in "human readable" form in m_TStr. If the TCP/IP sever function is active, the bytes in m_TStr are sent to the TCP/IP loopback address. At this point, various flags, strings and other variables are reinitialized for building the next packet.

In addition to the correlator, the decoded bit stream is fed into a byte assembler which includes bit-destuffing, and a bit-wise FCS calculator. The first fields in a packet contain the To and From address, and usually some repeater addresses. During byte assembly, the "end of the address field" bit is detected and the number of bytes in the address field is stored in AfC for later use. The packet bytes are assembled into a raw packet in string "Str", with byte values that CEditView does not like to print replaced with spaces (for a cleaner display look). And while the address field byte are coming in, the To, From, and the first two repeater addresses are loaded into their own string variables for building the human-readable packet string. Expecting the packet monitor to be used mostly for APRS, I have not tried to decode/display some of the fancier, little used parts of an AX.25 frame. This is another homework assignment for you if your interested.

At the bottom of the "for (i = 0; i < (int)(DXS.m_BufSize / 2); i++)" DSP calculation loop, data samples are loaded for graphical display by CScope according to the view and mode the user has chosen. Each time the "for" loop completes, m_TStr is tested to see if it is holding a packet. If so, the packet is loaded into CDSPView's m_PktStr string and a user-defined message is posted to trigger CDSPView into displaying it.

Simple TCP/IP Communications

When I started to write the packet monitoring program, I anticipated that driving WinAPRS through the PC's TCP/IP stack would be quite involved. It actually turned out to be quite simple.

MFC has a class called CAsyncSocket that allows a simple TCP/IP server function to be added with just a dozen or so lines of code. You will find the initialization code just below the variable declarations in CDSPThread::DSPLoop(). Note you need two socket objects, one to listen for a connection request and one to actually make the connection.

Conclusion

As expected, I have found writing the DSP part of an application much easier in MFC/VC++ than in the assembly language I used for DSP-93 applications 10 years ago. In addition to the high-level MFC/VC++ programming language, floating point calculations, built-in trig functions, graphical data displays and high CPU capacity greatly simplify DSP design and debugging. On the other hand, gaining some capability in MFC/VC++ Windows programming has taken much longer than I expected. PC performance is continuing to improve rapidly, along with the growing availability of high performance sound and video I/O cards. It will be interesting to see how Amateur Radio DSP takes advantage of these hardware improvements to further evolve over the next 10 years.

Notes

[1]F. Perkins, WB5IPM, "DSP-93 Programming Hints,", *Proceedings of the 14th ARRL Digital Communications Conference*, Arlington, Texas, September 1995, pp 97-100.

[2]R. McGregor, *Using C++* (Indianapolis, Indiana: QUE, 1998, ISBN 0-7897-1667-4).

[3]J. Prosise, *Programming Windows with MFC*, second edition (Redmond, Washington: Microsoft Press, 1998, ISBN 1-57231-695-0).

[4]R. Jones, *Introduction to MFC Programming with Visual C++* (Upper Saddle River, New Jersey: Prentice Hall, 2000, ISBN 0-13-016629-4).

[5]E. Kain, *The MFC Answer Book* (Boston, Massachusetts: Addison-Wesley, 1998, ISBN 0-201-18537-7).

[6]J. Bates and T. Tompkins, *Practical Visual C++ 6* (Indianapolis, Indiana: QUE, 1999, ISBN 0-7897-2142-2).

[7]D. Chapman, *SAMS Teach Yourself Visual C++ 6 in 21 Days* (Indianapolis, Indiana: SAMS, 1998, ISBN 0-672-31240-9), pp 495-520.

[8]J. Beveridge and R. Wiener, *Multithreading Applications is Win32* (Boston, Massachusetts: Addison-Wesley, 1997, ISBN 0-201-44234-5), pp 223-243.

[9]B. Bargen and P. Donnelly, *Inside DirectX* (Redmond, Washington: Microsoft Press, 1998, ISBN 1-57231-696-9), pp 203-280.

[10]Microsoft, *Visual C++ MFC Library Reference, Part 1* (Redmond, Washington: Microsoft Press, 1997, ISBN 1-57231-518-0).

[11]Microsoft, *Visual C++ MFC Library Reference, Part 2* (Redmond, Washington: Microsoft Press, 1997, ISBN 1-57231-519-9).

[12]T. Grandgent, "Tom's Spectrum Analyzer Source Code," available at www.grandgent.com.

[13]M. Wheatley, AE4JY, "WinPSK 1.2 Source Code and Technical Reference Manual," available at www.qsl.net/ae4jy.

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

[15]M. Sproul, KB2ICI, and K. Sproul, WU2Z, "WinAPRS: Windows Automatic Position Reporting System," *Proceedings of the 15th ARRL and TAPR Digital Communications Conference*, Seattle, Washington, September 1996, pp 130-135.

[16]M. Frerking, *Digital Signal Processing in Communication Systems* (New York, New York: Van Nostrand Reinhold, 1994, ISBN 0-442-01616-6).

[17]T. McDermott, N5EG, *Wireless Digital Communications: Design and Theory* (Tucson, Arizona: TAPR, 1996, ISBN 0-9644707-2-1), pp 162-165.