Sivan Toledo, 4X6IZ

School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel; **stoledo@tau.ac.il**

A High-Performance Sound-Card AX.25 Modem

The author leads us through the process of fixing several known problems with AX.25 modems, leading to a new software modem.

I had been running an APRS RF-to-internet gateway (an iGate) for a few months using a sound-card modem by Thomas Sailer, HB9JNX/AE4WA, (soundmodem).[1] The software modem caused various problems and I have not found a suitable replacement. Eventually, I decided to try to implement my own sound-card-based software modem. The results have been very good, in spite of the fact that I do not have much background in digital signal processing.

This article describes the problems with current AX.25 software modems, the design methodology I followed in implementing the new modem, and of course, the resulting software. The methodology is particularly important; it has allowed me to design and implement a high-performance modem with little background in digital processing and absolutely no background or experience in designing digital decoders.

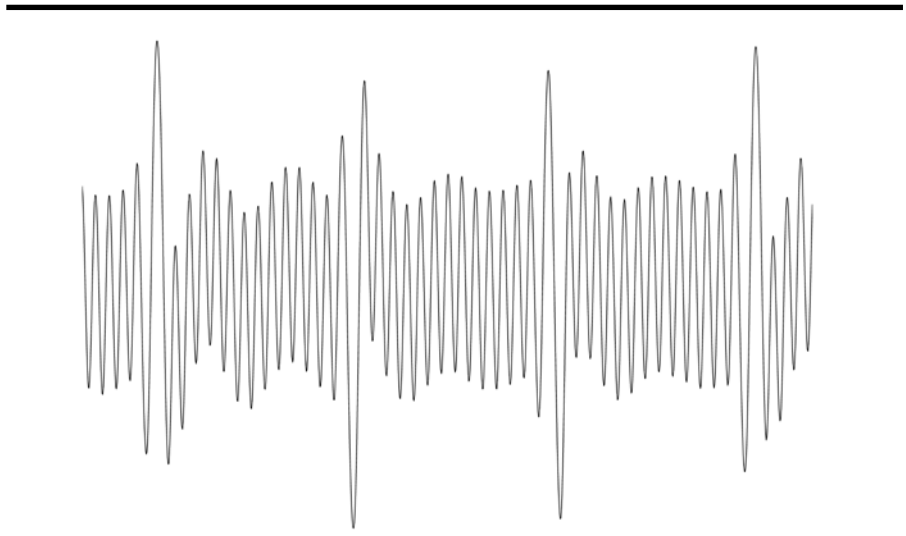

Figure 1 — This graph shows a piece of the audio I recorded from an AX.25 packet I received from a nearby station. The 2200 Hz tone amplitude is much lower than the 1200 Hz tone amplitude. Various software modems that I tried were unable to decode these packets.

Existing AX.25 Modems: Some Software, Some Hardware

VHF APRS uses AX.25 packets with 1200 baud audio frequency shift keying (AFSK) modulation. Binary ones are represented by a continuous tune, either 1200 or 2200 Hz, and zeros are represented by a switch from one of these tones to the other. This modulation scheme is based on the Bell 202 telephone modem standard from around 1980.

Three types of modems are in wide use in APRS systems today. One type uses a dedicated modem integrated circuit (IC), the mx614.[2] There is not a wide selection of 1200 baud modem ICs in production today. The IC is only responsible for generating the tones and for deciding which tone is present at any given time. In mx614-based modems, such as John Hansen's X-TNC and the OpenTracker series, the modem IC is connected to a microcontroller that determines the symbol timing, extracts the packet bit stream, and checks the packet for integrity. In many APRS systems the same microcontroller also performs other functions, such as converting GPS sentences to APRS messages.

In the second type of modem, the microcontroller is also used for modulation and demodulation, using its built-in analog-to-digital converter and a resistor-network digital-to-analog converter. Bob Ball's TNC, Byon Garrabrant's TinyTrak series, Scott Miller's OpenTracker series, and Robert Marshall's Arduino-based TNC all belong to this group.[3, 4, 5]

Modems in the third group run on a PC and rely on a sound card to transfer audio between the radio and the computer: Thomas Sailer's soundmodem and multimon, George Rossopoylos's AGWPE, Frank Perkins' AX.25-SCS, and Andrei Kopanchuk's recent modem (which I was not aware of when I started this project).[6, 7, 8, 9, 10]

Decoding and Interfacing Problems with Existing Software Modems

I faced two types of problems with the sound-card-based software modems that I tried. I started with AGWPE, connected through a homebrewed computer/radio interface to a Yaesu FT-857D. The software generated packets that nearby stations

[1]Notes appear on page 25.

were able to decode, but it could not decode packets from the same stations. I switched to AX.25-SCS, but the results were similar. I recorded the audio of several packets from stations that the modems could not decode and discovered that the amplitude of the 2200 Hz tone was much lower than the amplitude of the 1200 Hz tone, as you can see in Figure 1.

This turned out to be what caused difficulties to both AGWPE and AX.25-SCS. I switched to a different sound-card interface, which apparently does not attenuate high-frequencies as much, and both modems were able to decode many more packets. You may say that the first sound-card interface is simply not good enough, but AX.25 is modulated using *frequency* shift keying, so in theory, the demodulator should not be sensitive to amplitude variations.

It appears that soundmodem suffers from similar problems. When fed by an old Kenwood TR-2500, it decoded packets just fine, but when I switch to a Motorola radio (a PageTrac, in which the radio is the same as in the more common MaxTrac radios), it failed to decode most packets.

All of these modems need to interface to both a sound card and an AX.25 or APRS program. I also faced problems in this area. AX.25-SCS does not allow you to select a sound card, so I could not use it with an external high-quality sound card. Both of these programs, as well as the new UZ7HO modem, are *Windows*-only programs, so I could not use them in my *Linux*-based APRS iGate. Soundmodem does work under *Linux*, but interfacing it to Pete Loveall's *javAPRSSrvr* proved challenging.[11] I initially tried to interface the two programs using a virtual serial port, but *javAPRSSrvr* kept complaining that soundmodem was closing the serial port. I then interfaced them using the *Linux* kernel's support for AX.25 networking; this approach is somewhat more complicated, but it did work reliably. Either way, soundmodem must be started before *javAPRSSrvr*, otherwise *javAPRSSrvr* fails to find the virtual serial port or the AX.25 kernel interface.

In spite of these problems, I was able to find a working reliable configuration. A Kenwood TR-2500 fed soundmodem, which fed *javAPRSSrvr* through the *Linux* kernel's AX.25 support. This setup worked fine for a few months, but when I switched to the Motorola PageTrac and discovered that soundmodem would not decode packets received by it, I decided to try to design and implement a better AX.25 sound-card modem. I did not have much experience in digital-signal processing (DSP), but I decided to learn the necessary tools and to try to implement a good modem.

Pre-Emphasis and De-Emphasis

Before describing the modem and how I designed it, it is worth explaining why the amplitudes of the 1200 Hz and 2200 Hz tones in received packets often vary significantly. In FM voice communication, the transmitter pre-emphasizes high tones by 6 dB per octave. That is, it amplifies high audio frequencies more than it amplifies low frequencies, which results in wider deviation for high tones. To compensate, the receiver de-emphasizes the received audio by the same amount, to make the audio sound natural. This scheme originated in the use of phase modulators rather than frequency modulators, but it is still useful today because the de-emphasis in the receiver cuts down annoying high-frequency noise (hiss).

If all transmitters and receivers adhered precisely to the 6 dB/octave curve, and if all AX.25 modems were interfaced to the pre/de-emphasized audio path, we would not see any amplitude variations between the two tones. But some radios do not adhere precisely to the 6 dB/octave curve, and some radios interface the AX.25 modem directly to the modulator and/or discriminator, bypassing the pre/de-emphasis circuits. To make things even more complicated, some modems pre/de-emphasize the audio too, in an attempt to compensate for the radio.

The outcome of all of this is that no matter what you do at the receiving end, your AX.25 demodulator is going to see packets with amplitude variations between the tones. It is, therefore, important for the demodulator to be able to cope with such variations.

A Design and Evaluation Methodology

To design the modem and to evaluate how well it works, I used a methodology that is somewhat different from the methodology used in many other amateur projects.

One aspect of the methodology is the use of *Matlab* to do much of the design. I intended to write the modem in *Java* or *C* (I settled on *Java*), but for the initial prototyping I used *Matlab*, an interactive technical computing environment with excellent support for plotting graphs, for DSP, and for reading wav files (sound recordings). *Matlab* has other features, of course, but these three were important in this project.

I started by recording packets off the air, some from stations that I knew are hard to decode and some from stations that I knew were easy to decode. I recorded segments of about 5 minutes that I knew were likely to contain beacon packets (digipeaters tend to beacon at fixed intervals) and trimmed the audio using *Audacity*, a free cross-platform sound editor.[12]

I then started to develop the DSP algorithm, testing it on the recorded packets after every modification. The behavior of the code on the recorded packets helped me understand the DSP techniques that are used in modems, and it eventually showed that my demodulator was working. The ability to plot signals that are derived within the algorithm from the input audio, in particular, was crucial; it really helped me understand what works and what does not work. You will see some of these plots below, and I hope that you will agree that they do indeed provide insight as to how the algorithm works. The plots shown below are static; you cannot zoom in and out, select which time series appear on each plot, and so on. In *Matlab*, the plots are interactive, and hence even more useful.

My experience with AGWPE, AX.25-SCS, and soundmodem taught me that a modem that can decode a few packets from particular transmitter/receiver combinations can still fail on other transmitter/receiver combinations.

This brings us to the second aspect of the methodology: the use of Stephen Smith's APRS test CD.[13] This CD consists of several recordings of APRS traffic. The first two tracks on the CD are the most useful ones for evaluating modems. The first track is a recording of about 40 minutes of APRS traffic in Los Angeles, consisting of more than 900 packets from many different radios and modems. This track is a recording of the discriminator output of a radio, with no de-emphasis. The second track contains the same audio, but it is accurately de-emphasized by 6 dB/octave. If a modem works well on both tracks, it is likely to work well with most radios.

The test CD has been critical in optimizing the ability of my modem to decode packets. I ran the entire first two tracks through variations of my modem many times. From each such run, I would note the number of decoded packets from each track. This allowed me to test different filters and algorithms and to understand how different parameters of the algorithm affect its performance.

Using a real-world workload (sometimes called a benchmark) to monitor the evolution of a system as it is being optimized, and to compare systems is not new, of course. Stephen Smith made the test CD available precisely for this purpose. I learned of the CD from the website of Robert Marshall, who used it to compare his microcontroller-based AX.25 modem to several existing modems (his results, while perhaps not completely scientific, show that AGWPE and soundmodem indeed perform quite poorly).[14] The fact that the methodology is not new does not mean that it is widely used. It is not, usu-

ally because a suitable large and representative workload is not available. For example, Bob Ball, who designed his AX.25 modem before the test CD was available, described the performance of his modem in the following way: "My units routinely decode packets that are less than 1 S unit on my 2 meter radio from stations over 75 miles away." This is useful information, but quantitative information on a standard workload is much more informative.

The Design of the Modem

An AX.25 modem consists of a transmit path and a receive path that are almost unrelated (apart from decisions as to when to transmit, which are based on whether a packet is correctly being received). The transmit path currently consistists of an *encoder*, which encodes an AX.25 packet into a bit stream, and a *modulator*, which turns the bit stream into audio samples at a particular sample rate. Similarly, the receive part consist of a *demodulator* that transforms audio samples into bits, and a *decoder* that transforms the bits into packets (or drops them, if the packet was not received correctly in its entirety).

The encoder and decoder algorithms are completely described by the AX.25 specifications and they have been explained in many articles, so I will not describe them here. All that is important here is that the bit sequence that makes up a packet is *bit-stuffed*; if five ones appear in a row, a zero is inserted by the encoder (and dropped by the decoder) in order to ensure that zeros are not too rare. Another important piece of information is that *flag* bytes (hexadecimal 7E) surround each packet, and they are not bit stuffed, which means that if the decoder sees six ones in a row, it must be either a flag or just noise.

The modulator transforms the bit-stuffed sequence into audio. This is done using a *phase-continuous digital resonator* that runs at either 1200 Hz or 2200 Hz. The resonator has one state variable, an angle (between 0 and 2 π). To produce a sample at 1200 Hz, the resonator outputs sin (α) and advances α by 2 π / (f_S / 1200), where f_S is the sampling frequency. To produce a sample at 2200 Hz, the resonator outputs sin (α) and advances α by 2 π / (f_S / 2200). This is pretty simple. The demodulator is more interesting.

Time-Domain Filtering and Emphasizing

The processing of incoming signal samples starts with a filter that filters out some of the energy in frequencies outside the 1200 to 2200 Hz range. The same filter can also emphasize the 2200 Hz tone by 6 dB relative to the 1200 Hz tone; this is an option that the modem uses sometimes, but not always; more on that later.

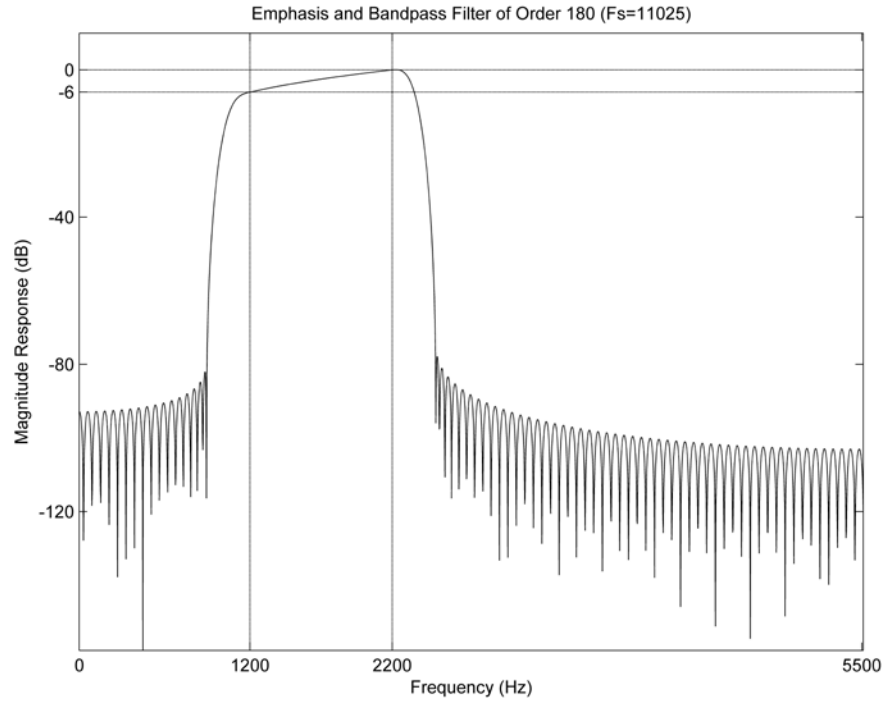

Figure 2 — Here is the response of the 900 to 2500 Hz high order (180) bandpass filter I created in *Matlab*. Note the 6 dB ramp up of the response across the passband, so there is no gain or attenuation at 2200 Hz.

Filtering out frequencies below 1200 Hz and above 2200 Hz is theoretically useless. I say theoretically because the filter is useless only if received noise obeys certain theoretical assumptions. But this filtering is cheap, harmless (even theoretically), and was found to be useful in some hardware-based modems, so I included it anyway.

We must not filter away energy in between 1200 Hz and 2200 Hz. During transitions from one tone to the other the signal contains energy in intermediate frequencies. If we filter these in-between frequencies, the filtered signal won't be able to switch from one tone to the other and will get stuck in one of the tones. I was initially not aware of this and learned this lesson the hard way: I produced a filter that essentially passed only 1200 Hz and 2200 Hz, and discovered that it strips away all the information in the signal!

The filter is an FIR filter that the modem applies in the time domain, by convolving the filter coefficients with incoming samples. I construct the filter in *Matlab* using the firls function, which constructs a filter whose frequency response matches a given response as well as possible given the filter order. The specification I have firls called for a complete attenuation of frequencies below 900 Hz and above 2500 Hz, with a 6 dB ramp-up from 1200 Hz to 2200 Hz, with no attenuation and no gain at 2200 Hz. Figure 2 shows what the frequency response of the filter looks like when we specify a high-order filter.

Applying high-order filters is expensive, however, and experiments showed that going to very high-orders does not improve the demodulator's performance (this is probably an outcome of the theoretical result I mentioned earlier). In practice, the demodulator uses filters whose order is twice the number of samples per symbol (that is, per $1/1200$ samples). The graph of Figure 3 shows the response of the actual filter that we use at f_S = 11,025. Compared to the order-180 filter, it looks fairly crude, but it still works well, as we'll see below. The response has a significant variation within the 1200 to 2200 Hz passband, but it still emphasizes the high tone by about 6 dB relative to the low tone.

Figure 4 shows a signal recorded off the air before and after this filtering. The sampling rate here is 48,000 samples per second, to give a high resolution. What we see is the flag (0×7E byte) just before the packet and the first data bits of the packet itself. In this case the original signal is fairly clean, so the filtered version is pretty close to the original, except for some attenuation and the elimination of dc. The filtering introduces a time delay.

Detecting the Tones Using Correlations

After filtering the signal, we try to detect which of the two tones was transmitted at every sample point. We do this by correlating the received samples with synthetic signals

that the demodulator generates at the two tone frequencies over a period of one symbol.

Let us examine this computation in some more details. The demodulator generates f_S / 1200 samples of a sine and f_S / 1200 samples of a cosine, and stores them in two arrays, s and c. When processing a new received sample, the demodulator puts it in an array together with the previous (f_S / 1200) – 1 samples, in increasing time order. This is done efficiently by using the array as a cyclic buffer. Now the demodulator loops over the f_S / 1200 input samples and multiplies each one by the corresponding sine sample and the corresponding cosine sample. The products for the sine are added up and the products for the cosine are added up, the two sums are squared, the squares are added, and finally the square root of that sum is taken. This is the correlation value for the sample just received.

Formally, the correlation is an inner product of the received signal with a complex exponential function, but there is an easy way to understand why the correlation detects tones without resorting to complex numbers. If the incoming signal is at the frequency we correlate with and at the same phase as our synthetic sine, the sum of squares for the sine products will give a high value and the cosine will give a zero. If the phase corresponds to the cosine, the situation will be reversed, but the sum of the two sums of squares will be large either way. If the incoming signal's phase is somewhere in between, both the sine and the cosine will contribute to the correlation, and the correlation will still be high. The use of both a sine and a cosine essentially compensates for the fact that we have no idea what the phase of the received signal is. If the received signal's frequency does not match closely that of the sine and the cosine, the sample-by-sample products will produce both positive and negative numbers that will tend to cancel out; the overall correlation will be low. Figure 5 shows the two correlation signals for the signal shown in Figure 4. The correlation with 1200 Hz tends to be high when the 1200 Hz tone is present and low otherwise, and the correlation with 2200 Hz behaves just the opposite. They appear quite noisy, but nonetheless they contain very useful information.

Decision Making: Which Tone is More Dominant?

To decode the AX.25 bit stream, we need to find transitions between the two tones. This requires us to decide which tone is more dominant at every sample. As we can see from the computed correlations displayed in Figure 5, the correlation signal can be quite noisy; it is not easy to decide whether a particular tone is present or not. (The noisiness

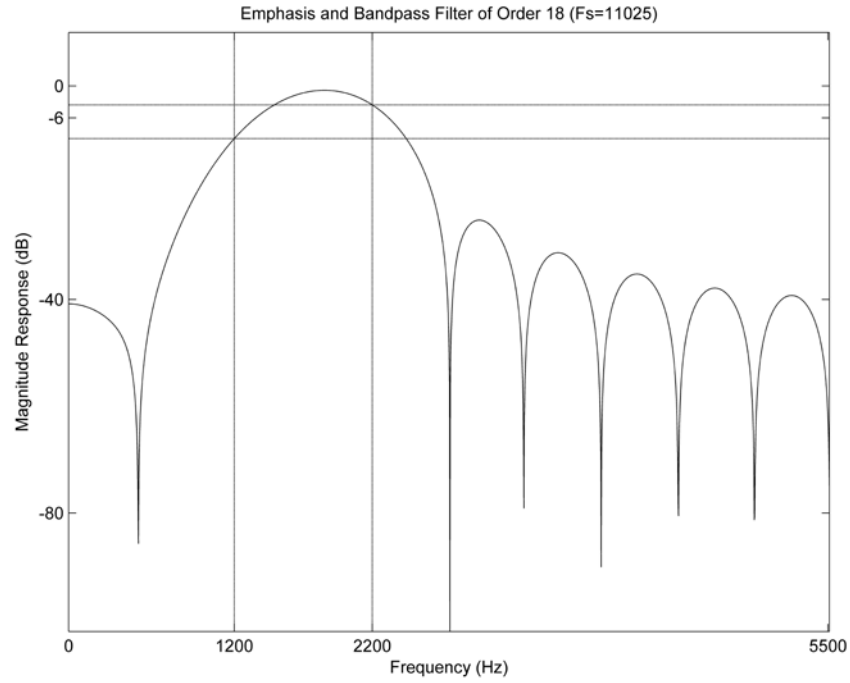

Figure 3 — This graph shows the response of a more practical low order (18) bandpass filter. It does not look nearly as good as the 180 order filter, and has a significant variation across the passband, it still emphasizes the high tone by about 6 dB.

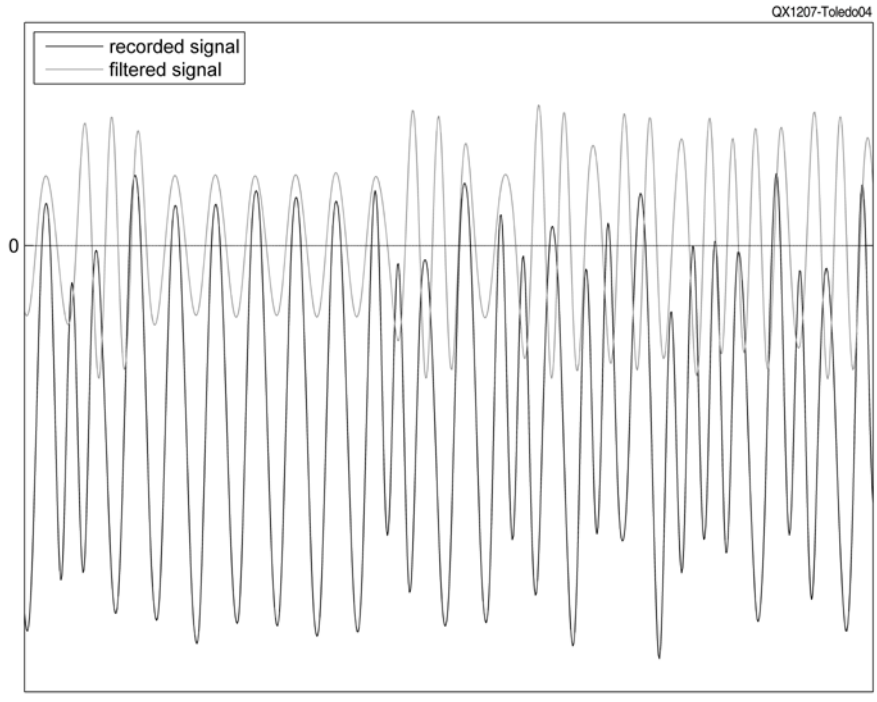

Figure 4 — Here is a graph of a signal recorded off the air, along with the filtered version after applying the bandpass filter of Figure 3. The filter has eliminated a dc component of the signal, but is a close representation of the original signal. You can also see that the filter has introduced a slight time delay.

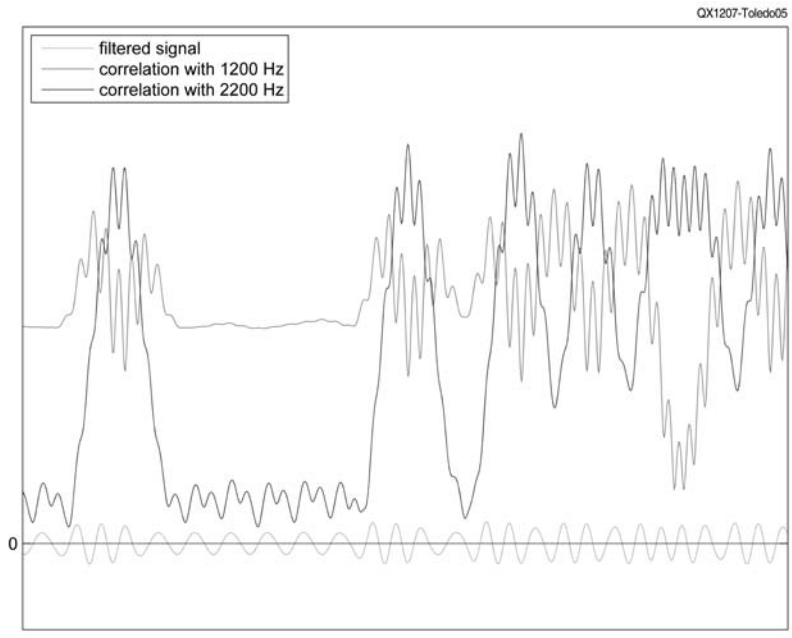

Figure 5 — Here we see the filtered signal, centered on the 0 line, along with the correlation signals for the 2200 Hz tone (highest when that tone is present and lowest when it is not) and the 1200 Hz tone (high when that tone is present, but not as low when it is not).

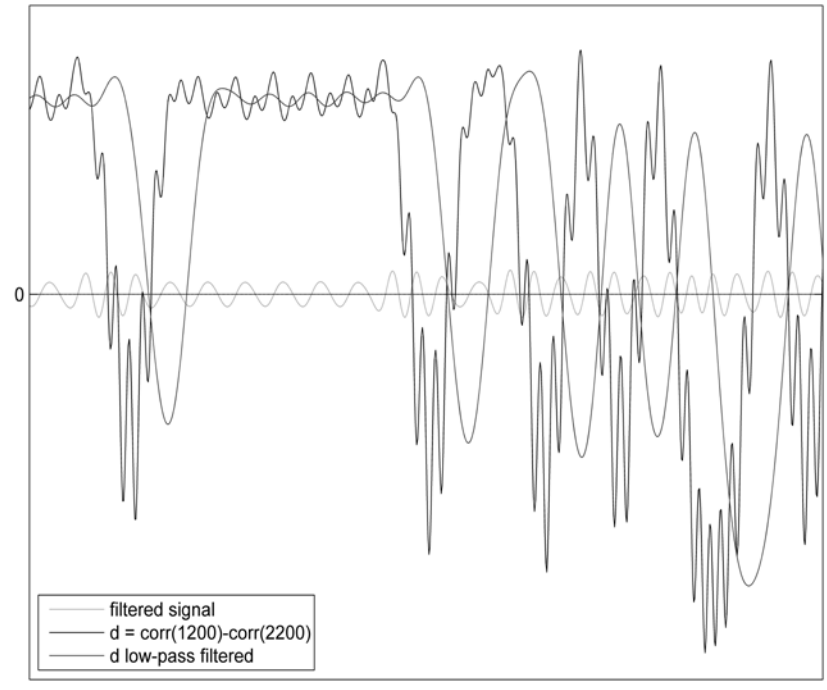

Figure 6 — This graph shows the filtered signal, centered on the 0 line, and the difference between the high and low tone correlation (the noisiest signal). After applying a low pass filter to the difference between tone correlations, we get the smoothed signal.

of the correlations is clearly evident in the 1200 Hz correlation; the 2200 Hz correlation is cleaner.) Deciding *which tone is stronger* is easier, however. We simply compare the two correlation signals and estimate which tone is present by selecting the stronger correlation. In the graph of Figure 6 we show the difference between the correlations; a positive value means that the 1200 Hz tone is stronger and a negative value means the 2200 Hz tone is stronger.

Like the correlation signals, the correlation-difference signal is also noisy. This makes it difficult to find the transitions between the two tones, because the high-frequency noise in the difference signal often causes several zero crossings to appear in quick succession. Fortunately, we know that tone transitions occur at most once every $^{1}/_{1200}$ of a second. Therefore, we can pass the correlation-difference signal through a 1200 Hz low-pass filter. The graph of Figure 6 shows that as expected, the low-pass filtering maintains the overall shape of the signal but removes the high-frequency noise and the spurious zero crossings that they generate.

The low pass filters that the demodulator uses have been constructed using the fir1 function in *Matlab*, which uses a Hamming window to construct the filter. The filter order is the same as that of the time-domain band-pass and emphasis filter, twice the number of samples in a symbol period.

Recovering the Bit Stream

At this point the demodulator can reliably determine the timing of transitions between the two tones. It is time to recover the AX.25 bit stream. The first step is to determine how many symbol periods separate consecutive transitions. The graph in Figure 7 shows the lengths of these periods in our signal.

Due to various potential sources of noise, the periods between estimated tone-transition times are not exactly integers, but they are close. The number of samples per symbol in this signal is 40, so an error of one sample is equivalent to an error of 0.025 symbol periods; we see that the errors in our processing of this signal are up to two samples.

At lower sample rates there are fewer samples per symbol. At 9600 samples per second, we have only 8 samples per symbol, but the method still works reliably. At 8000 samples per second, however, rounding errors become common and the method often fails. To avoid these errors at the 8000 samples/second rate, we interpolate the signal to 16000 samples/second and then decode it; this works reliably.

Detecting a transition 1 symbol period after another means that we have decoded a zero bit in the AX.25 bit stream; the modu-

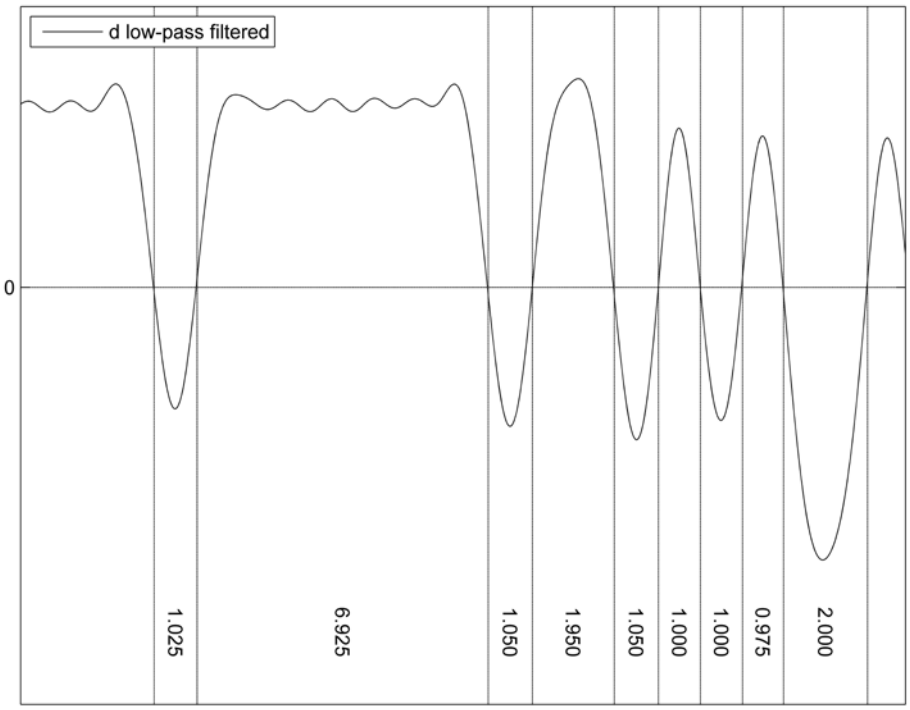

Figure 7 — Here is the low pass filtered signal from Figure 6. The symbol periods can easily be identified, so we can begin decoding the packet. Note that the longest time between transitions corresponds to 7 symbol periods, which represents a flag byte.

Table 1

		Track 1 (No De-emphasis)					*Track 2 (6 dB De-emphasis)*				
		8	*16*	*32*	*64*	*128*	*8*	*16*	*32*	*64*	*128*
Flat	11025	946	941	943	948	948	824	859	716	809	657
Filter	44100	436	937	967	961	961	77	418	891	856	760
Emphasis	11025	949	728	504	528	555	844	951	958	957	955
Filter	44100	439	938	966	927	524	76	539	895	945	959

lator sent a symbol for one period and then switched tones to indicate the zero. A transition after 2 periods means that we have decoded a one followed by a zero. Three periods decode into 2 ones and a zero, and so on up to 5 periods. When we detect a transition after 6 periods, we have decoded 5 ones but without a zero after them; a zero was stuffed into the bit stream to generate a transition so that the demodulator can stay synchronized with the modulator; the zero is not part of the AX.25 bit stream. When we detect a transition after 7 periods, we have decoded a flag byte, which might delineate a packet but is not part of a packet; we see such a flag in Figure 7 (the 6.925 periods get rounded to 7). A transition after more than 7.5 periods or less than half a period indicates a decoding error and causes the decoder to abort the current packet and to search for the flag that delineates the next packet.

Once we have decoded a supposed packet delineated by two flag bytes, the decoder checks the checksum value that is transmitted as the last two bytes of the packet. If the checksum is correct (it should be a specific function of the rest of the packet), the packet is considered valid and is passed to the client of the modem (typically an APRS program); otherwise the packet is discarded.

Optimizing the Demodulator

The demodulator's performance, both in terms of the amount of computation required and the ability to decode noisy and/or de-emphasized packets depends on the design of its two filters and on the sampling rate. The filters can be designed in many different ways: different filter orders, different cutoff and transition frequencies, and different filter-design algorithms. Understanding how all of these parameters affect the demodulator and how they interact is not easy, certainly not to people with limited background in DSP.

To address this issue, I performed many systematic experiments in which I tested different parameters on the first two tracks in the test CD. Table 1 shows some of the results of these experiments (it does not reflect exactly the performance of the final demodulator). The table shows the number of packets decoded from each track in two sample rates (11025 and 44100), with two types of time-domain filters (one with the same magnitude response at 1200 and 2200 Hz and the other with 6 dB emphasis), and at 5 filter orders (8 to 128).

We learn a lot from this table and from others like it. We see that filter orders that are much shorter than the number of samples per symbol lead to very poor performance. Filters of order 8 are fine for 11,025 samples/second but are terrible for 44,100. We see that a filter that is matched to the emphasis in the radio helps, but we also see that this is particularly true for filters of very high order; shorter filters often deliver good results in both tracks (for example, filter order 32 or 64 at 44100 samples/second).

Striking Twice to Hit Once

The realization that a flat filter is best on some signals and that an emphasized filter is best on others caused me some worries. I tried to come up with strategies to choose the correct filter; some of them were quite complex.

I eventually realized that I can avoid the choice altogether. I can feed the audio samples to two demodulator algorithms running in parallel. Sometimes both of them will decode a packet (and then a simple duplicate removal algorithm discards one of them). At other times only one will be able to decode a packet.

Running each audio sample through two copies of the demodulator is twice as expensive as running only one demodulator. On very weak computers (such as smart phones) this may be significant, but on desktops, laptops, and servers — even weak ones — the demodulator is cheap enough to run two copies in parallel. On a 1.6 GHz Intel Atom 330 computer (a relatively weak and slow CPU) running two demodulators in parallel on 11,025 samples / second audio uses only 8% of the CPU power.

Table 2 shows how many test-CD packets are decoded by the strike-twice-hit-once demodulator and by the two single-filter demodulators. We can see that using two demodulators with different filters is beneficial even compared to using a filter that matches the de-emphasis setting in the receiver; when there is a mismatch, the benefit is dramatic.

Software Design, Status, and Availability

One of the first choices that I made early

Table 2

	Track 1 (No De-emphasis)			Track 2 (6 dB De-emphasis)		
	Flat Filter	*Emphasized Filter*	*Both Filters*	*Flat Filter*	*Emphasized Filter*	*Both Filters*
8000	960	939	966	854	950	954
9600	964	686	966	854	957	958
44100	961	917	962	881	959	964

in this project was to completely decouple the signal-processing routines from the interface routines, mainly so that the modem could be used in different operating systems and environments. By interface routine I mean both the interface to the sound card and the interface to the modem client, typically an APRS program. The decision was partially based on an earlier experience trying to port parts of soundmodem from the *Linux*/*Windows* environment for which it was written to a microcontroller-based tracker. It was hard. Soundmodem does use a modular architecture, but still the separation between the modules was not clean enough to make separation easy.

The separation between the DSP routines and all the other software routine in the new modem is clean, which makes porting the DSP routines to new Java platforms and client environments easy.

The DSP routines are currently integrated into four different software environments. Three of these environments were written by me and I did the integration; somebody else authored the forth and integrated the modem into it. These environments are:

1) A command-line program to test the modem and to measure its performance. This program can generate packets into a sound card and can decode packets from either a sound card or a wav file.

2) *javAPRS*, an APRS iGate and digipeater software written by Pete Loveall.[15] *javAPRS* comes with interfaces to various types of hardware and software modems (serial TNCs, *Linux* kernel AX.25 support and AGWPE). I added support for my new modem, which can now run as an integral part of *javAPRS*.

3) An AGWPE emulator. This program does not implement the entire AGWPE TCP/IP protocol (which is not publicly documented, to the best of my knowledge), but it implements enough of it to support APRS client programs like APRSISCE.[16]

4) APRSdroid, an APRS client for Android phones and tablets written by Georg Lucas.[17] Georg integrated my modem into APRSdroid, reporting that the initial integration effort was easy and that the interface code (which did not include error checking at that point) consisted of only 50 lines. The audio processing mechanism in Android is completely different from the mechanism in *Java* running under *Windows*, *Linux*, and *Mac OS*; therefore, the ease of integration shows that the DSP routines in the modem are indeed well separated from the audio processing routines.

The modem has been running under *javAPRS* around the clock for a few months now as both an iGate and as a SatGate; both copies of *javAPRS* and the modem run on the same computer using two different sound cards (the internal one and an inexpensive USB sound card) and two radios. The iGate gates packets to and from RF; the SatGate only gates packets from RF to the internet. The modem has been tested under *Linux*, *Windows*, and *Android*.

The new modem is freely available along with a user manual under an open-source license.[18]

Sivan Toledo is Professor of Computer Science at Tel-Aviv University. He holds BSc and MSc degrees from Tel-Aviv University and a PhD from the Massachusetts Institute of Technology, where he was also Visiting Associate Professor in 2007-2009. He was licensed in 1982.

Notes

[1] You can find more information and download this software at **www.baycom.org/~tom/ham/soundmodem/**.

[2] James A. Mitrenga, N9ART, "An MX614 Packet Modem," *QST*, Jan 2000, pp 44-46.

[3] Bob Ball, WB8WGA, "An Inexpensive Terminal Node Controller for Packet Radio," *QEX*, Mar/Apr 2005, pp 16-25.

[4] See the Byonics website for more information about Byon Garrabrant's TinyTrak modems: **www.byonics.com**.

[5] Scott Miller's OpenTracker series of modems are available from Argent Data Systems: **https://www.argentdata.com/**.

[6] You can download Thomas Sailer's soundmodem software at: **www.baycom.org/~tom/ham/soundmodem/**.

[7] Thomas Sailer's multimon modem software is available at: **www.baycom.org/~tom/ham/linux/multimon.html**.

[8] You can find Georg Rossopoylos' AGWPE modem software at: **www.sv2agw.com/ham/agwpe.htm**.

[9] Frank Perkins, WB5IPM. "DSP Programming using DirectSound and MFC/VC++," Proceedings of the 22nd ARRL and TAPR Digital Communications Conference, Hartford, Connecticut, September 2003, pp 140-149.

[10] Andrei Kopanchuk's software modem is available for download at: **http://uz7.ho.ua**.

[11] For more information about Pete Loveall's javAPRSSrvr software modem see: **groups.yahoo.com/group/javaprssrvr/**.

[12] You can download *Audacity* for free at **audacity.sourceforge.net/**

[13] For more information about Stephen Smith's test CD, see his website: **wa8lmf.net/TNCtest/index.htm**.

[14] There is more information about Robert Marshall's microntroller-based modem on his website. **http://sites.google.com/site/ki4mcw/Home/arduino-tnc**.

[15] For more information about Pete Loveall's software modem see: **http://groups.yahoo.com/group/javaprssrvr/**.

[16] For more information about the APRSISCE APRS client program, see: **http://groups.yahoo.com/group/APRSISCE/**.

[17] Georg Lucas has written an APRS client for Android smart phones: **http://aprsdroid.org/**.

[18] You can download the software for the modem described in this article from my website: **https://github.com/sivantoledo**. The software version as of the publication date of this article is also available for download from the ARRL QEX files website. Go to **www.arrl.org/qexfiles** and look for the file **7x12_Toledo.zip**. QEX