An HF Frequency-Division Multiplex (FDM) Modem

Steven Sampson, K5OKC

Introduction

A popular new digital voice mode has been FreeDV designed by David Rowe, VK5DGR. The modem and speech vocoder were programmed in the C++ and C languages. However, my language of choice is usually Java, which is not your typical Digital Signal Processing (DSP) language. It may have a slower execution speed, but it does allow for rapid prototype development using the Netbeans Integrated Development Environment (IDE).

As far as digital voice is concerned, any conversion speed faster than 10 ms is like a billionaire at a garage sale. The power of the typical desktop computer today, allows the Java double-precision math implementation to easily run within this timing requirement. Working with the sound card is also quite easy with this language.

This paper is about an FDM modem implementation, but in order to test the modem, I also translated the Codec2 speech vocoder into Java. That left a simple GUI design to finish the prototype.

I've been interested in the Amateur Radio data modes for most of my time in the hobby, and for the FDM modem I wanted it to work in data, as well as voice. I did not want to get bogged-down in protocol requirements; however, so this modem design should be considered a Layer 1 design, as there is no addressing or handshaking at this stage. Not even a Cyclic Redundancy Check (CRC).

For voice mode, any digital errors will be processed by our ears and brains, as we don't have the luxury of waiting for a data packet to be error checked and corrected before converting to analog audio. For the text mode, we can visually see any typed errors, and then request a resend as needed. Later we can add CRC, and ARQ protocols and make the modes more rugged.

Modem Description

The FDM modem is a very interesting design. In the day when Orthogonal FDM modems are common-place, the FDM seems rather nostalgic. Sometimes good enough, is good enough. It consists of sixteen (16) Differential Quadrature Phase Shift (DQPSK) subbands, or sometimes called carriers, separated by 75 Hz. Each one of these subbands sends two bits as one of four phases, for a total of thirty-two (32) bits. There is also a synchronization subband that uses Differential Binary Shift Key (DBPSK) to send a pilot signal we can use to send sixty-four (64) bits as odd and even 32 bit halves, and also provides a center frequency to track with. The 64 bits being the magic number required by the speech vocoder, as it needs that many bits to represent the digital voice. The modem operates at 50 baud. That is, the 16 subands are sent 50 times a second and transmitting 32 bits per baud. Thus, for sending both halves, the effective rate is 25 baud, or 64 bits at 25 times a second. One speech frame is processed every 40 ms, which is the Codec2 frame rate. This then provides us with 1600 bits/sec, and fits in about 1.3 kHz of bandwidth. The modem audio is also centered on 1500 Hz, which puts it in the middle of a typical audio passband.

Rewriting the modem and codec into Java, I was able to create two objects that we can use in future applications. By instantiating these two objects, and calling their exported methods, developers can

use the objects and ignore their inner design. The objects are just black-boxes.

Expanding the Design

The object of this exercise; however, is to expand the basic modem design a bit. We want to add a text mode that works as simple as the voice mode. That means we need some kind of Mode bit to toggle between these two modes. A simple solution is to focus on our 64 bit requirement. So our mode switch must be encoded into this size frame. Therefore I designed a Header Frame to have a 32 bit synchronization (Sync) word (1ACFFC1D used in the satellite industry), followed by a 32 bit Packet Definition (PD) word. This PD is sub-divided into a four bit Version word, a four bit Mode word, an eight bit Rollover Counter, and finally a sixteen bit Sequence Number.

With that, we can create a receiver State Engine that waits for the Sync word to finish, and then decodes the following PD word to setup for the next frame decode. In order to fit all of this into the 64 bit frame sending rate, I have designed the modem to send nine voice or text frames at a time. A total of ten 64 bit frames that is called the superframe, and results in a total of 640 bits.

In the text mode, we will send nine of these eight byte frames per superframe (72 bytes total), and in the voice mode we will send nine 64 bit vocoder frames. As you can see, we will start losing our real-time audio by 40 ms every 400 ms superframe, as we need to borrow one of the modems frames to use as the header control word. Because this is a half-duplex design, it isn't critical that we have zero delay in the voice or text transmission. Adding this Header Frame does reduce our effective modem bit rate. Instead of 1600 bit/sec of pure data being sent, we now send 1536 bit/sec. The 64 bits being lost to overhead.

Scrambling the Bits

One of my design objectives was to run an experiment, where I scramble the data bits in order to reduce the peak to average ratio you can see in the frequency spectrum. The highs can be really high, and the lows can be really low relative to each PSK subband. With a scrambler, we can make it look more like a constant amplitude noise. This is the theory anyway. In order to scramble the data, I created an RC4 object. The data to be sent is first scrambled, and then sent to the modem. Since we are using a fixed (unchangeable) key, this is not considered encryption or hiding information. There is no secret, thus the data remains plaintext. This may seem a bit of legal semantics, but it is legal in North America (at least) to scramble bits in a modem.

The RC4 algorithm has been modified to include a random Sequence Number, and a Rollover Counter. This then allows us to send the scrambled data in blocks, that if corrupted, do not affect any other block. To decode the scrambled block, we send the unscrambled PD word to the distant receiver. The receiver then plugs these values into its RC4 decoder, and decodes the scrambled superframe of data. Worse case, if we lose the PD word through corruption, we only have to throw out nine voice or text frames (360 ms of data). You might miss a vowel here and there.

Since all stations use the same key, the RC4 pseudo-random output bytes will always start with the same number, and produce the same sequence. This is how the receiver at the far end knows how to synchronize and decode the data.

The problem is, our modem only sends 72 bytes in our superframe, and the same random numbers would be output in each of these. The period (360 ms) is too small. Thus, one method is to add a 16

bit Sequence Number to the random number generator. Then broadcast this sequence number to the receivers, who can then synchronize their own generators. Since this is a generic algorithm, we also add an 8 bit Rollover Counter. When the sequence number overflows, we increment the rollover counter, and we can then have very large superframes with the same algorithm.

Now we can send a different 72 random bytes every superframe, and everyone stays in sync. Different groups of random numbers are exclusive-or with the data each superframe. The only hazard being that the ionosphere or interference may corrupt our transmission, in which case we have to discard everything until the next superframe.

In our simple HF modem, the period is quite small, and the sequence number and rollover counter fields are grossly miss-sized, but since we have to fill-up 64 bits with something anyway, there would be no advantage in optimizing their sizes, and we keep the generic features of the RC4 object class.

Receiving the Data

At the receiver end, we wait for a Sync word, and extract the sequence and rollover values. A function in the RC4 then allows us to prime the generator with these values, and we then exclusive-or the random bytes with the received data producing our unscrambled vocoder bytes. We will receive nine vocoder frames, and produce 360 ms of voice audio when we send data to the Codec2 speech vocoder.

Adding Functionality

The nice thing about having Mode bits, is that we can now send text or data in various formats. In this example, 72 bytes.

The mode switch doesn't have to be a physical switch. We could also send text or data before and after the voice. Another example might be sending your GPS position along with an encoded symbol, such as HQ, Mobile, Fixed, etc.

Improvement Areas

The current vocoder design uses 52 bits of data for the voice, and 12 bits for the FEC. However, in this modified design, the FEC is in the wrong place. Any error in receiving the scrambled bits will produce garbage, and the FEC bits won't help correct anything after the fact. An alternative vocoder is the 1600 bit/sec version. It outputs 64 bits of voice data, and no FEC. Fits easily where the 1300 + 300 FEC fit before.

Alternative Scrambling

The above algorithm was brought about because of the use of two or more modes. If only the voice alone, or data alone mode was to be used, we could simplify the scrambling algorithm. Like the 9600 baud modems of the last century, we can use a short polynomial to scramble the bits. A good choice is the 23 bit polynomial X^23+X^18+1 which is used in many modem designs. I have created two versions, one designed for 2 bit pairs, and another for 4 bit quads. The QPSK bit pairs are sent to the function on transmit, and the scrambled bit pair is sent to the modulator. The reverse on receive. With this type of scrambler, you can synchronize the modems with a long header. Sending a bunch of sync characters, before sending the actual data. Also, sending some more syncs at the end to flush the 23 bit polynomial register.

This algorithm, like the other, is prone to errors in receiving the bits, so the FEC added to the 1300 mode, is after the fact, and you will generally have more than three bad bits (the FEC maximum). Here again, you will have to throw out the bad frame, but the frame is small enough to be on the

order of a vowel, or small enough that you can re-send the packet if using ARQ.

Conclusion

As you can see, all of these options can be fun to play with. This is a fairly simple prototype, that builds on the Open Source software developed by David Rowe, that he has been engineered for our general use. As a code base, I find it interesting to expand and play with for purely hobby reasons. Expanding the number of subbands for use on VHF would also be interesting, and then you could use much larger superframes.

Availability

The experimental code is available at my github cloud: https://github.com/k5okc