Toward New Link Layer Protocols

Phil Karn, KA9Q

ABSTRACT

This paper describes an experimental new link layer protocol for amateur packet radio use. It extends my earlier MACA scheme for dealing with hidden terminals by incorporating a powerful forward error correction (FEC) scheme: convolutional coding with sequential decoding. The resulting hybrid protocol combines the best of the FEC and retransmission (ARQ) techniques. It should perform very well in the presence of hidden terminals, noise and interference, particularly pulsed radar QRM like that often found on the 70cm amateur band.

1. Introduction

AX.25 [1] is now 12 years old. Although it has become the universal standard link level protocol for amateur packet radio, it is widely recognized as being far from optimal. The experience of the past decade plus significant advances in the computer technology now available to the radio amateur suggest that we look at much more sophisticated alternatives. These techniques have existed for many years[1], but only now have powerful modern PCs brought them within easy reach of the average radio amateur.

This paper gives a brief overview of forward error correction techniques and then describes the author's work in progress to develop a new general purpose link level protocol suitable for amateur packet radio use at speeds up to about 100 kb/sec. This protocol combines the features of my earlier MACA (Multiple Access with Collision Avoidance) access scheme [17] with forward error correction (convolutional encoding with sequential decoding performed in software) in what the literature calls a **Hybrid Type II ARQ/FEC protocol**. [15,19] This protocol should perform well over channels with error rates up to 10%, far beyond the capabilities of AX.25. It was originally motivated by, and should be especially well suited for, channels with radar interference. With the right modem, it should also perform very well over analog satellite transponders like those on AO-13.

2. Computer technology - 1994 vs 1982

In the late 1970s and early 1980s, the best computer that most radio amateurs could hope to own included a 4 or 6 MHz Z-80 running CP/M, 240 KB 8" floppy drives, 64KB RAM, and either a "dumb terminal" or small memory mapped video display. The introduction of the IBM PC in the early 1980s upped these capabilities considerably, particularly in address space, but these machines were quite expensive when first introduced.

Since the original PC was introduced, there have been several more generations of Intel microprocessors: the 80286, first used in the PC/AT; the i386; i486; and now the Pentium. The faster 486 CPUs are nearly 100 times faster than the 8088 used in the original IBM PC.

Other measures of personal computer system performance have increased even more dramatically. It is not at all uncommon to find amateur systems with 16 megabytes of RAM (256 times the 64KB maximum with CP/M) and hundreds of megabytes or even gigabytes of hard disk (thousands of times the capacity, and hundreds of times the speed, of the 8" floppies used with CP/M). These machines rival the high-end engineering workstations of just a few years ago.

[1] Many of the fundamental references cited in this paper date from the 1970s or even earlier.

And best of all, these machines have plummeted in price even as their capabilities have grown. The 8088 has long been obsolete; the 286 more recently so. Mother boards with the 386 chip are now selling for less than a hundred dollars, a sure sign that they too will soon be obsolete.

3. What modern computers make possible

When AX.25 was developed, it was reasonable to emphasize simplicity over functionality and performance. DARPA had already experimented with much more sophisticated and powerful packet radio protocols, error correction and signal processing techniques, [18,21,22] but theirs was a government funded research program with (by amateur standards) very deep pockets. Amateurs needed a practical standard they could afford, even if it didn't work very well.[2]

But many of the sophisticated techniques used in the DARPA project are now within our easy reach. Thanks to their wide data paths and high speed caches, chips like the 386 and 486 are especially well suited to CPU-intensive operations such as forward error correction. Yet amateurs have been slow to exploit these capabilities by updating their protocols.

Amateur packet radio urgently needs FEC. A common rule of thumb is that the packet loss rate should not exceed 1% for good performance with pure ARQ[3] protocols like LAPB[4] or TCP.[5] The typical amateur packet radio channel far exceeds this. Causes include non-optimal modem designs and radio interfaces, poor signal-to-noise ratios, and interference.

FEC is especially effective against radar. We share our 70cm amateur band with military radar, and in the coastal cities the periodic audio whine of a Navy radar is a familiar sound.[6]

4. Forward Error Correction - background

Forward Error Correction (FEC) has been around for quite some time. It essentially began when Claude Shannon's revolutionary 1948 paper launched the field of information theory. [2]

Error correction is a very deep and often highly mathematical subject, about which dozens of textbooks and thousands of journal papers have been written. I could not possibly include a complete discussion in this paper. However, the basic principles are easy enough to understand.

Shannon proved that it is possible to send data over even a noisy channel with an arbitrarily low error rate, as long as the data rate is less than the channel capacity, defined by the famous formula

$$C = B \log_2 (1 + \frac{S}{N})$$

Unfortunately, Shannon did not show *how* one could achieve this capacity, only that it was theoretically possible. Practical systems have only begun to approach the Shannon limit, although some have come remarkably close (the Voyager spacecraft down link operated at about 50% of the Shannon capacity of its channel).

The basic idea behind FEC is to add redundancy to the data being sent so that some number of channel errors can be corrected, up to a limit. The encoded data are referred to as **symbols**, to distinguish them from the original user data bits.

Compare this with the more familiar ARQ scheme, where redundancy (e.g., a CRC) is added to the data only to *detect* errors; if even one error occurs, the receiver discards the entire block of received data and waits for a retransmission. If this happens often, a significant amount of channel capacity and transmitter energy is wasted.

[2] The military also discovered the same thing, with the ironic result that amateur packet radio technology found its way into several military applications.

[3] automatic request-repeat: (re)send each packet until an acknowledgement is received

[4] the connection-oriented part of AX.25, borrowed from X.25 Level 2

[5] the Internet's connection-oriented transport level protocol

[6] Much of the early work on FEC was prompted by the Navy's desire for reliable shipboard communication links that could tolerate local radar QRM. My present interest was initially prompted by the heavy radar QRM we often experience to our 70cm 56kb/s modems here in San Diego.

With FEC, a receiver can use *all* of the received signal energy; it doesn't have to throw any of it away. This actually allows one to reduce the transmitter power required for a given user data throughput. This reduction is called the **coding gain** of the code. Depending on the particular code, its implementation, the modem in use and the amount of redundant information added, coding gains can range from 1-8dB or even more against additive white gaussian noise (AWGN, e.g., the thermal noise of a preamp and antenna system).

White noise is actually a worst case for FEC; certain codes can provide dramatically higher gains against non-white noise such as radar pulses, and against short deep fades. Coding gains of 40-50dB or more against radar QRM are easily achievable, assuming that the receiver recovers quickly after each radar pulse.[7] The error rate in a radar-QRMed digital system depends primarily on the user data rate and the radar pulse duration and repetition rate. It is essentially independent of radar signal level over a wide range. For example, radar pulses 10-20 microseconds long are comparable to a bit time at 56kb/s (17.8 microseconds). If the radar pulse power exceeds the desired signal power minus the required S/N demodulator margin, a data bit will be corrupted. However, if the receiver recovers rapidly after each pulse, the bits sent between pulses will arrive unharmed no matter how much stronger the radar gets. At a repetition rate of 400 Hz, only every 140th data bit will be clobbered. This is an error rate of .7%, which is *easily* handled with just a modest amount of FEC. On the other hand, without coding communication would be almost impossible unless you could increase transmitter power to completely overcome the radar, and this is quite often impractical.

A similar situation exists with fading. If the fades are relatively short, coding can easily handle the situation no matter how deep the fades are.[8] The alternative is to increase transmitter power until there's enough margin even in the deepest fade, which may again be impractical. FEC essentially "smears" individual data bits over time on the channel so it is not necessary to receive an entire message, only enough of it to permit the decoder to fill in the missing spots from what it did get.

There is no free lunch, however; coding gains necessarily come at the expense of increased bandwidth. For example, the code I am currently using in my experiments provides a 4-dB coding gain against AWGN at the expense of doubling the required bandwidth by sending two encoded symbols for each user data bit. That is, if I have a 10 kHz channel capable of carrying 10 kb/sec with 100 watts of transmitter power, then my code enables me to send the same 10 kb/sec with only about 40 watts of power, but I need 20 kHz of bandwidth to do it. Or I could stay in my original 10 kHz channel, reduce my data rate to 5 kb/sec, and reduce my transmitter power to only 20 watts.[9]

Because all FEC schemes add some amount of redundancy to the user data being transmitted over the channel, it is important to distinguish between the user data rate and the encoded symbol rate as sent over the channel. Since it is the user data rate that ultimately counts, the literature uses the parameter $\frac{E_b}{N_0}$ rather than "signal to noise ratio" to define the performance of a coding system and modem. E_b is the received energy per user data bit; N_0 is the received noise spectral density.

5. Types of FEC

5.1. Block Codes

The earliest error-correcting codes inspired by Shannon's work were Hamming's "block" codes [3]. As the name implies, a block code operates on a fixed amount of data that depends on the particular code. Hamming's code has been used on several amateur spacecraft to correct radiation-induced memory errors; the memory word makes a natural code "block". This code can generally correct one error in each 8-bit byte of memory. Another popular block code, the Golay code, adds 11 redundant bits to 12 data bits to produce 23 bits for transmission. This is referred to as a (23,12) block code. This code can correct any combination of

[7] A pulse blanker is important to keep the radar energy out of the receiver AGC and IF filters. It can also directly aid the FEC process, as explained later.

[8] A radar QRMed channel with a pulse blanker at the receiver is really just a channel with infinitely deep fades during each pulse.

[9] The extra 3dB of power reduction in the latter case is *not* part of the coding gain; it results directly from the 50% reduction in user data throughput.

three or fewer errors in its transmitted block of 23 bits. [4]

Another important block code (out of many) are the Reed-Solomon codes, used in the compact disc and elsewhere [4,5]. Reed and Solomon actually defined a whole family of codes. Reed-Solomon codes can operate on multi-bit symbols (as opposed to binary symbols) and can be easily constructed with relatively large block sizes. These two characteristics together provide an excellent burst-error-correcting capability, which is especially useful when combined or **concatenated** with other error detecting and correcting codes. The Clover II HF modem [12,13,14] designed by Ray Petit, W7GHM, makes extensive use of Reed-Solomon coding.

5.2. Convolutional Codes

Another class of error correcting codes operates on an arbitrary stream of bits, rather than fixed-sized blocks. These are the convolutional codes, sometimes called "tree" codes. In a more general form they are known as "trellis" codes and are found in modern dialup telephone modems (e.g., V.32 and V.32bis).

Convolutional codes are easily generated in hardware with a shift register, two or more exclusive-OR "parity trees" and a multiplexor. These operations are also easily implemented in software. The length of the shift register defines an important parameter called the "constraint length" K of the code. The larger K is, the better the code will perform (subject to the limitations of the decoding process as discussed below).

Because a convolutional code operates on a continuous stream, they are specified by the ratio of the input data and output symbol rates. For example, an encoder that produces two encoded symbols for each user data bit is known as a "rate 1/2" coder, often abbreviated to "r=1/2".

Convolutional codes have several interesting properties. They are especially easy to generate and provide excellent performance for a given amount of complexity. They are well suited to varying amounts of data, e.g., variable length data packets. They are also readily adapted to **soft decision** decoding.[10] This uses symbol quality indications from the modem to aid the decoding process. For example, instead of slicing the demodulator output to binary 0 or 1 with a comparator, one uses a A/D converter to indicate the relative quality of each 0 and 1.[11]

When properly implemented, soft decision decoding yields a $\frac{2}{\pi}$ (2dB) performance improvement over 1-bit or **hard decision** decoding in the presence of AWGN. The code I’m using would therefore provide a 6-dB coding gain were I to use full-blown soft decision decoding, but this will have to wait for modems (most likely DSP-based) that provide soft decision samples.

My decoder currently accepts "unknown" as a received symbol value in addition to "one" and "zero". This 3-level **binary erasure channel** is a good model for a radar-QRMed channel where symbols are erased by radar pulses. Making this erasure information available to a decoder can dramatically improve its performance. For example, without erasure information the rate 1/2 code I’m using can, with considerable effort, barely correct a stream of symbols when the symbol error rate reaches 10%. However, if the modem can tell the decoder *which* symbols have been trashed by radar pulses (perhaps by running the receiver’s noise blanker gate line to the decoder), then the decoder will continue to function even when nearly 50% of the received symbols have been erased -- i.e., when almost all of the redundant information added by the encoder has been removed by the channel.

A 3-level decoder is also useful for **puncturing**, another useful technique easily applied to convolutional codes. This involves simply not transmitting some fraction of the encoded symbols and substituting erasures in their place at the decoder.[12] Puncturing makes it easy to vary the coding rate (the amount of redundant information added to each user data bit) in accordance with changing channel conditions without having to change the encoder or decoder. This is important because the redundant information added by a coder is merely useless overhead when the channel conditions are good enough to not require the code’s full error correcting capability.

[10] It is possible to "soften" the decoding of block codes, but with considerably more effort. [6]

[11] A simple comparator is really a 1-bit A/D converter, so we are just increasing the resolution of the A/D converter we already have.

[12] Obviously the receiver has to know which symbols are not sent, otherwise it will get very confused.

6. Sequential vs Viterbi decoding

There are two general classes of decoders for convolutional codes: **sequential** and **maximum-likelihood** (Viterbi). The sequential decoder is the older of the two, dating from the late 1950s. [7] The Fano algorithm for the sequential decoding [8] dates from the early 1960s and is still in use with few changes.

Viterbi first proposed his algorithm in the late 1960s. [9] It has since become the most popular technique for decoding convolutional codes, primarily because of the availability of high speed VLSI hardware decoders. [10] The Voyager spacecraft down link mentioned earlier uses convolutional encoding with Viterbi decoding, sometimes in conjunction with a Reed-Solomon block coder.

Both the sequential and maximum-likelihood decoders work by regenerating the data stream that, when passed through a local copy of the encoder, most closely matches the received encoded symbol sequence. Because the received symbols usually contain errors, the match is seldom exact; but as long as the error rate is not too high the decoder will reproduce the original user data without any errors. This is possible because the effect of any particular user data bit is spread over many encoded channel symbols. For example, in the K=32 rate 1/2 code I use, each data bit affects 64 encoded symbols. Even when several symbols are lost, there is usually enough information left in the remaining symbols to reconstruct all of the original data bits. The coding gain of a convolutional code depends on the constraint length, K; the larger the better.

The main difference between the sequential and maximum-likelihood decoders is the way they regenerate the original data sequence that is fed through the decoder's local copy of the encoder for comparison with the received symbol stream. Both use what might be called "intelligent brute force", as each tests and discards many data sequences that don't work out. This requirement for "brute force" CPU crunching is why convolutional decoding is just now becoming practical for amateur use at reasonable speeds.

The Viterbi decoder explores every possible data sequence in parallel, discarding at each step those that can't possibly be correct. These parallel operations are well suited to hardware VLSI implementation. The number of data sequences that a Viterbi decoder explores in parallel is 2^K. Increasing K by 1 doubles the work that the decoder must do to recover each data bit. Viterbi decoders typically work with K=7 or K=9; larger values are very rare.

The sequential decoder, on the other hand, explores only one data sequence at a time, so its work factor is almost independent of K. As long as the sequence being tested produces symbols reasonably close to those being received, it keeps on going. When the decoder "gets into trouble", it backs up and methodically searches increasingly dissimilar data sequences until it eventually finds one that gets it back onto the right track. Much larger values of K are typically used with sequential decoders than with Viterbi decoders. My code uses K=32, a convenient value on modern microprocessors like the 386 and 486 with 32 bit registers.

Since larger values of K give better performance, this seems to tip the scales in favor of sequential decoding. However, the Viterbi decoder always operates at a constant rate, while the speed of the sequential decoder is a random function that depends strongly on the channel error rate; the higher the error rate, the greater and more unpredictable the decoding time becomes. At sufficiently high error rates, the decoding time of a sequential decoder becomes unbounded. [11] Any practical sequential decoder must therefore have a timer to keep it from running "forever" if it is inadvertently fed garbage or an extremely noisy packet. This makes the sequential decoder less attractive than the Viterbi decoder for real-time applications with strict delay limits such as full duplex digital voice.

On the other hand, the *average* decoding time of a sequential decoder is often less than the fixed decoding time of a software Viterbi decoder running on the same CPU. This is important if a) you've decided to implement your decoder in software to avoid the cost of a dedicated VLSI Viterbi decoder, and b) you have a non-real-time packet application such as file transfer, where it's the average decoding speed that counts.

An interesting property of sequential decoders with large values of K is that the probability of uncorrected errors becomes very small. It is much more likely that the decoder will time out first. This may make it unnecessary to include a CRC or other error detecting code to ensure that the decoded data is correct; the decoder timer takes the place of a CRC check. We don't actually avoid the overhead of the CRC, though. A convolutional coder requires a **tail** of zeros at the end of every packet to return the coder to its starting state and to allow every user data bit to influence the full number of channel symbols. The tail length must be at least K-1 bits, or 31 for my code (I use 32 just to keep things in round numbers).

Whether the inherent error detecting ability of my K=32 sequential decoder provided by its timeout mechanism is comparable to that of a 32-bit CRC is an interesting question. I don't know yet.

So to recap, Viterbi decoding is usually preferable when the application requires a constant decoding delay and dedicated VLSI hardware decoders are available. Sequential decoding has the edge when a variable decoding delay is tolerable and a software implementation is required (e.g., to minimize cost to the radio amateur who already has a PC). For these reasons I have chosen sequential decoding for my experimental protocol.[13]

7. Interleaving

Convolutional decoders work best when the symbol errors they correct are evenly distributed throughout the transmission. This is the case on channels limited by AWGN (thermal noise). However, errors caused by momentary interference and fading often occur in bursts. These can cause sequential decoders to get into trouble since the work factor goes up exponentially with the length of the burst.

Interleaving is the standard approach to this problem. The symbols from the encoder are rearranged in time before transmission and put back into their original order before decoding. Interleaving doesn't remove any errors, it simply scatters them in case they were adjacent on the channel.

Several classes of interleaving schemes exist. The particular interleaving scheme I've chosen for my protocol uses **address bit reversal**. Each symbol address is written in binary in the usual way with the high order bit on the left. Then the bits are reversed right-to-left, forming a new number. For example, the sequence

0 1 2 3 4 5 6 7

when bit-reversed becomes

0 4 2 6 1 5 3 7

Note that all the even numbers are in the first half of the block, and all the odd numbers in the second half. Note also that when these numbers are again bit-reversed, the original sequence reappears.

Since address bit reversal only works when you have 2^N numbers, I need some way to extend this scheme to arbitrary length packets. To facilitate this, I first pad each data packet out to a multiple of 32 bits (64 symbols for my rate 1/2 code). Then I write the symbols as a 2-dimensional matrix with 64 rows, first going vertically down the columns and using as many columns as I need. Here is an example with 192 symbols, numbered 0 through 191:

0	64	128
1	65	129
2	66	130
.	.	.
.	.	.
63	127	191

Then I interchange the rows by bit-reversing the row addresses:

0	64	128
32	96	160
16	80	144
.	.	.
.	.	.
63	127	191

Now I actually transmit the symbols by transmitting horizontally across each row. The transmitted symbol sequence is therefore

0, 64, 128, 32, 96, 160, 16, 80, 144, . . . , 63, 127, 191

[13] The DARPA packet radio project also chose convolutional coding with sequential decoding.

At the receiver I reverse the process, restoring the symbols to their original order. Note how adjacent symbols from the encoder are always widely separated in time when they go over the channel; in particular, note how all of the even numbered symbols appear in the first half of the transmission and the odd numbered symbols in the second half.

8. Variable Rate Puncturing by Interleaving

I chose this particular interleaving scheme because it makes variable-rate code puncturing especially easy. Suppose I transmit only the first 32 of the 64 rows. This covers all of the even-numbered symbols in the stream; I've sent every other symbol. Since there are twice as many symbols as data bits, the effective code rate is 1/2 divided by 1/2, or 1 (i.e., no redundant information, and no ability to correct errors). Now suppose we also send the 33rd row. This gives us 1/2 divided by 33/64, which is 32/33. This "high rate" code cannot correct as many errors as the original rate 1/2 code, but if this is good enough for the channel, we can avoid sending the other 30 rows. On the other hand, if the channel is poor, we simply send enough rows to lower the effective code rate until decoding is possible.[14] Adding the 34th row gives us a code rate of 16/17, and so on up to all 64 rows, which returns us to a rate 1/2 code.

Nothing says we have to send all of the rows in a single transmission. We could start by sending just the first 32 rows (no redundancy) and attempting to decode the packet with a tight timeout. Although we cannot actually correct any errors with only 32 rows, if any do occur the decoder will "get stuck" in the tail and time out, thus indicating that errors exist. If this happens, we can send additional rows until the receiver is finally able to decode the packet. In this way we send only as much redundancy as the channel currently requires. This is the idea behind the Type II Hybrid ARQ/FEC scheme mentioned earlier: use the power of FEC to deal with channel errors, but use the adaptability of ARQ to adjust the FEC overhead to that actually required by the channel. [15, 16, 20, 24]

What if the receiver cannot decode the packet even after we send all the rows? One possibility is to send the whole thing again and to use **code combining** at the receiver to add the two transmissions before decoding. [23] For example, if we send all of the symbols belonging to a rate 1/2 code twice, we have effectively switched to a rate 1/4 code. Although this particular rate 1/4 code provides no additional coding gain over the original rate 1/2 code (since the retransmissions are identical), adding the two transmissions increases the total received energy (and $\frac{E_b}{N_0}$) by 3dB.[15] Of course, this comes at the expense of halving the user data rate. We could take this scheme even further by adding more than two transmissions but we are eventually limited by the packet synchronization mechanism discussed in the next section.

This scheme can provide some of the benefits of soft decision decoding even when only hard decision samples are available from the modem. For example, the DARPA packet radio could combine two hard-decision copies of a packet by erasing those symbols that disagreed between the two transmissions. [18] Symbols that agreed were left unchanged. As we've already seen, a sequential decoder has a much easier time dealing with erasures than with errors.

9. Synchronization

Any packet protocol needs something to reliably flag the beginning of a packet. HDLC uses the 8-bit value 7E (hex) for this purpose, but since we want to operate reliably over very noisy channels this is not acceptable. As anyone who has ever operated a packet station with the squelch open and "PASSALL ON" knows, you don't have to wait very long for a 7E to appear by chance in random noise. And what if a packet is actually present, but one or more bits in the flag is in error? The entire packet would be missed. We *could* accept flags with, say, at most one bit in error but this would make the false alarm problem even worse.

The solution is to use a longer flag or **sync vector**. The longer the sync vector, the easier it is to reliably detect real packets with errors while rejecting false alarms (triggering the sync detector on random noise). I

[14] Sending additional rows also increases the total transmitted energy. This increases the $\frac{E_b}{N_0}$ ratio, which also aids decoding.

[15] Pactor uses **Memory ARQ**, which is essentially a combining scheme on uncoded data. FEC and interleaving could improve Pactor's performance considerably without any increase in bandwidth.

have chosen a 64-bit sync vector for my protocol that consists of a 63-bit PN sequence (generated by a 5-stage shift register with feedback) augmented by an extra 0 to make 64 bits. The receiver correlates the incoming symbol stream against a local copy of the sync vector, and it declares synchronization whenever they match with 13 or fewer errors. This allows the detector to work reliably up to a channel error probability of about 20%, well above the error correcting capability of the rate 1/2 convolutional code (about 10%).[16] Yet the sync vector is so long that the probability of random noise triggering the detector is quite small.

10. Scrambling

Several higher speed packet radio modems, e.g., the K9NG/G3RUH 9600 bps FSK modems and the WA4DSY 56 kb/s modem, use scrambling to ensure a sufficiently high bit transition density to allow the demodulator to recover clock regardless of the user's data sequence. However, the self-synchronizing descramblers they use have an unfortunate property: error propagation. Each channel error produces a characteristic pattern of several closely spaced data bit errors that depends on the particular polynomial being used. Without FEC this is of little consequence since even a single bit error is enough to ruin a packet, so extra errors can't make things worse. But with FEC, we don't want the modem to do anything to make the decoder's job harder. So we must disable the scrambling and descrambling functions.

This leaves us with the problem that scrambling was originally intended to solve: how can we ensure a good transition density on the channel no matter what the user sends? It turns out that the solution is relatively simple: we scramble the user's data ourselves, but we don't use a self-synchronizing descrambler. We use a fixed PN sequence that is started from a known point at the beginning of the packet and allowed to "free run" for the length of the packet. This type of scrambling doesn't propagate errors, but it does require independent synchronization -- which we already have from the sync vector mechanism just described.

It may turn out that current modem clock recovery mechanisms are inadequate with FEC. FEC can operate with a $\frac{E_s}{N_0}$[17] far lower than that required to produce good data without coding, and it's entirely possible that existing modem clock recovery circuits won't work on these weak signals. Other approaches may be necessary.

One possible approach is to perform the sync vector correlation function in a DSP modem on raw A/D input samples at some integer multiple of the incoming data rate. When the correlator output peaks, we can then start blindly counting off the appropriate number of A/D samples between each received symbol. If the transmitter and receiver clocks are closely matched in frequency and the packets aren't too long, this should provide reasonably accurate symbol timing for the entire packet without having to extract clock from those (noisy) symbols. If the clocks are too far apart for this to work, another possibility would be to buffer all of the raw A/D samples in the packet and post-process them looking for the sampling frequency that produces the best eye pattern for the packet as a whole.

11. Protocol Headers

The discussion of variable rate code puncturing assumed that we have had a reliable way to control it. But the control information can also be corrupted by channel errors. How can we deal with this? By always using full FEC coding for the packet header, regardless of the coding rate in use for the user data portion of the packet. Since the header is (hopefully) small compared to the user data, the overhead incurred by this is (hopefully) also small. To do this, though, we have to be very selective about what goes into the header.

The header in my protocol is currently 16 bytes (128 data bits or 256 encoded symbols) long. It contains the following information: source address, destination address, frame type, transmission length, previous frame error count, and coder tail.

[16] This provides some margin to allow successful code combining, which could allow us to operate above a 10% symbol error rate.

[17] symbol energy to noise spectral density ratio, as opposed to $\frac{E_b}{N_0}$. For a rate 1/2 code, $\frac{E_s}{N_0}$ is 3 dB less than $\frac{E_b}{N_0}$.

The coder tail is all zeros. As mentioned earlier, it is required by our use of a convolutional coder. Because we restart the coding process between the header and the data portion of the packet, each portion needs its own coder tail.

The source and destination addresses consist of the station call signs and SSIDs as in AX.25, but the call signs are more efficiently coded. Since there are only 36 legal characters in a call sign (the 26 letters in the English alphabet plus the ten decimal digits) there is no need to spend an entire 8-bit byte on each one. If we add ’space’ as a 37th character and use radix-37 encoding, we can encode into 32 bits any legal call sign up to 6 characters long[18]. Add 4 bits for an SSID and the complete address fits into 36 bits, as compared with 52 for AX.25.

The frame types are as follows:

- Request to Send (RTS)
- Clear to Send (CTS)
- User Data
- Negative Acknowledgement (NAK)
- Positive Acknowledgement (ACK)

The RTS tells the receiver of the sender’s intention to send a certain amount of data. The receiver responds with a CTS that echoes this length. This tells the sender to go ahead with the actual data transmission, and it also has the important side effect of telling anyone else on the channel to remain quiet for the appropriate amount of time.

If the receiver is able to decode the transmission (the sequential decoder completes without timing out), it returns an ACK to the sender. This tells the sender that it may proceed to the next block of data. Alternatively, if the timer expires before the sequential decoder finishes, the receiver returns a NAK. The NAK confirms to the sender that its transmission was received, but with too many errors. The sender then sends additional rows from the interleaver output, which the receiver combines with those symbols already received. If the receiver is still unable to decode the packet, the cycle repeats with additional NAKs and data packets containing additional rows of symbols until the receiver finally succeeds and returns an ACK. At this point the sender can continue to the next block of data.

Note that the NAK, like the CTS, also holds off other stations from transmitting so that they do not interfere. In fact, I may eventually merge the NAK and CTS messages into a single type, with the CTS simply being the special case of a NAK sent before any data symbols have been received.

It also goes without saying that should the sender receive neither an ACK or a NAK in a reasonable time, it must retransmit its last frame. This can only occur if the channel was so poor that the heavily coded header could not be decoded, or perhaps even the sync vector was missed. If this is a temporary condition, then a retransmission will get things moving again.

The previous frame error count field is used in ACK packets to let the sender know how many errors were detected and corrected in the last packet. The sender can use this information to aid in deciding how many of the interleaver symbol rows to send up front in its next transmission, without having to wait for the receiver to ask for them with NAK messages. This helps minimize modem transmit-receive cycles when channel conditions are fairly constant. At the moment I actually have two previous frame error count fields: one for the last header received, and one (in ACK packets only) for the last data field received.

12. Status and Open Questions

Since sequential decoding is the heart of this protocol and by far the biggest consumer of CPU cycles, I have spent most of my time to date working on my implementation of the Fano algorithm in C. It now runs quite well on the 486-50. At the moment, this decoder can, on average, easily keep up with a 56 kilosymbol/s stream (e.g., from a WA4DSY modem) as long as the symbol error rate is less than about 2%. More

[18] Some might like to see room for longer call signs. However, except for the rare special event station, all amateur radio call signs have been 6 characters or less and are likely to remain so for quite some time. In my opinion, the legal requirement for IDs with prefixes during reciprocal operation (e.g., "W6/GB1AAA") is better met with a special ID frame every 10 minutes than by requiring everyone to make room for them in every packet header. I’m willing to be persuaded otherwise. Remember that this is still an experimental protocol...

errors could be tolerated at a lower channel speed, with a faster CPU, or with a better optimizing compiler, but this performance is already good enough to be quite useful.

The correlator that searches for the sync vector is another potentially CPU-intensive task. Although correlation is simpler than sequential decoding, the decoder runs only when a packet has actually been received, while the correlator may have to process a continuous symbol stream from the modem if it has no carrier-detect squelch.[19] So it is still desirable to make the correlator run as fast as possible in order to free up the maximum amount of CPU for other tasks. I have implemented a correlator in assembler that runs at several hundred kilosymbols per second, with most of the time spent in the function that returns the next received symbol.

I have not yet finished the complete protocol, however. When I do, I will have plenty of work left. I need to answer the following questions:

1. Is a rate 1/2 code strong enough, particularly for the packet header? Should I consider using a rate 1/3 or even lower coding for maximum robustness?
2. Should I consider a block code (e.g., Golay) for the packet header in order to eliminate the need for a coder tail?
3. What strategy should the sender use to decide how many rows (out of the 64 available) should be sent in any given transmission? How can I make best use of recent history (especially the receiver's observed error rate indication) to send just the required amount of redundancy for each packet in as few transmissions as possible?
4. Is 64 interleaving rows optimum? Is the whole interleaving scheme optimum?
5. What is the optimum packet size for transmission? Should transmissions be large to decrease header and modem turnaround overhead, or should they be small to decrease the chances of a sequential decoding timeout and the resulting need to transmit additional redundancy?
6. Is relying on a decoder timeout sufficient to detect errors when a packet has been punctured back to rate 1 (no redundancy), or is a true CRC still required?
7. Does the MACA algorithm work well in the presence of stations too far away to reliably decode CTS messages, but close enough to cause harmful interference? Does FEC help this problem by improving the capture effect?
8. Can I make a sequential decoder that works well on soft decision samples such as those that might be produced by a PSK modem implemented in DSP, or is the sequential decoder's well-known sensitivity to incorrect soft-decision metrics a serious stumbling block? Can I compute the metrics on the fly according to observed noise levels to mitigate this problem?

9 Will the clock recovery circuits in existing amateur packet radio modems turn out to be the limiting factor instead of the error correcting capability of the code?

and last but not least,

10. Will the average amateur be willing to use this stuff?

13. Credits

I would like to thank several people for their advice and assistance: Franklin Antonio N6NKF and Klein Gilhousen WT6G, for sharing their considerable expertise in the practical application of convolutional coding; Paul Williamson KB5MU and Bob McGwier N4HY for their insights and especially their patience in listening to my ravings; Andy Demartini KC2FF of Digital Radio Systems Inc, for his donation of two DRSI PCPA Type 1 interface cards; and Skip Hansen WB6YMH for his clever trick of turning a Zilog SCC into a simple and dumb serial/parallel converter.

[19] Good carrier detect circuits are already hard to build, and it will probably be almost impossible to make them operate quickly and reliably at the much lower $\frac{E_s}{N_0}$ ratios usable with FEC. And recall that one of the principles behind MACA is that carrier detect is essentially worthless in a hidden-terminal environment anyway.

14. References

[1] Terry Fox WB4JFI, ed, "AX.25 Amateur Packet-Radio Link-Layer Protocol Version 2.0, October 1984", ARRL. (Updates earlier versions dating from 1982).

[2] C. E. Shannon, "A Mathematical Theory of Communication", Bell System Technical Journal, vol. 27, July/October 1948, pp 379-423.

[3] Richard. W. Hamming, "Error Detecting and Error Correcting Codes," Bell System Technical Journal, April 1950.

[4] Shu Lin and Daniel J. Costello, Jr., "Error Control Coding: Fundamentals and Applications", Prentice Hall, 1983 (ISBN 0-13-283796-X).

[5] Ken C. Pohlmann, "Principles of Digital Audio", second edition, Sams, 1990 (ISBN 0-672-22634-0).

[6] David Chase, "A Class of Algorithms for Decoding Block Codes With Channel Measurement Information", IEEE Transactions on Information Theory, Vol IT-18, No 1, January 1972.

[7] J. M. Wozencraft, "Sequential Decoding for Reliable Communications", Res. Lab. of Electronics, MIT, Cambridge, Mass., Technical Rept. 325; 1957.

[8] Robert M. Fano, "A Heuristic Discussion of Probabilistic Decoding", IEEE Transactions on Information Theory, April 1963, pages 64-74.

[9] Andrew J. Viterbi, "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm", IEEE Transactions on Information Theory, Vol IT-13, No 2, April 1967.

[10] Qualcomm, Inc., data sheets for Q1650, Q0256, and Q1401 Viterbi Decoder ICs.

[11] Irwin Mark Jacobs, "Sequential Decoding for Efficient Communication from Deep Space", IEEE Transactions on Communication Technology, Vol COM-15, No 4, August 1967, p. 492.

[12] Raymond C. Petit W7GHM, "The Cloverleaf Performance-Oriented HF Data Communication System", Proceedings of the 9th ARRL/CRRL Amateur Radio Computer Networking Conference", London Ontario Canada, September 22, 1990, p 191.

[13] Raymond C. Petit W7GHM, "Clover-II: A Technical Overview", Proceedings of the 10th ARRL Amateur Radio Computer Networking Conference", San Jose CA, September 27-29, 1991, p. 125.

[14] Bill Henry K9GWT and Raymond C. Petit W7GHM, "HF Radio Data Communication", Communication Quarterly, Vol 2 No 2 Spring 1992, p. 11.

[15] Samir Kallel and David Haccoun, "Generalized Type II Hybrid ARQ Scheme Using Punctured Convolutional Coding", IEEE Transactions on Communications, Vol 38, No 11, November 1990, p. 1938.

[16] Samir Kallel and David Haccoun, "Sequential Decoding with ARQ and Code Combining: A Robust Hybrid FEC/ARQ System", IEEE Transactions on Communications, Vol 36, No 7, July 1988, p. 773.

[17] Phil Karn, "MACA - A New Channel Access Method for Packet Radio", Proceedings of the 9th ARRL/CRRL Amateur Radio Computer Networking Conference", London Ontario Canada, September 22, 1990, p. 134.

[18] R. E. Kahn, S. A. Gronemeyer, J. Burchfiel and R. C. Kunzelman, "Advances in Packet Radio Technology", Proceedings of the IEEE, November 1978, p. 1468.

[19] Shu Lin and Philip S. Yu, "A Hybrid ARQ Scheme with Parity Retransmission for Error Control of Satellite Channels", IEEE Transactions on Communications, Vol COM-30, No 7, July 1982, p. 1701.

[20] David M. Mandelbaum, "An Adaptive-Feedback Coding Scheme Using Incremental Redundancy", IEEE Transactions on Information Theory, May 1974, p. 388.

[21] Nachum Shacham, "Performance of ARQ with Sequential Decoding Over One-Hop and Two-Hop Radio Links", IEEE Transactions on Communications, Vol COM-31, No 10, October 1983, p. 1172.

[22] Special issue on packet radio networks, Proceedings of the IEEE, January 1987.

[23] David Chase, "Code Combining - A Maximum-Likelihood Decoding Approach for Combining an Arbitrary Number of Noisy Packets", IEEE Transactions on Communications, Vol COM-33, No 5, May 1985, p. 385.

[24] John J. Metzner, "Improvements in Block-Retransmission Schemes", IEEE Transactions on Communications, Vol COM-27, No 2, February 1979, p. 524.