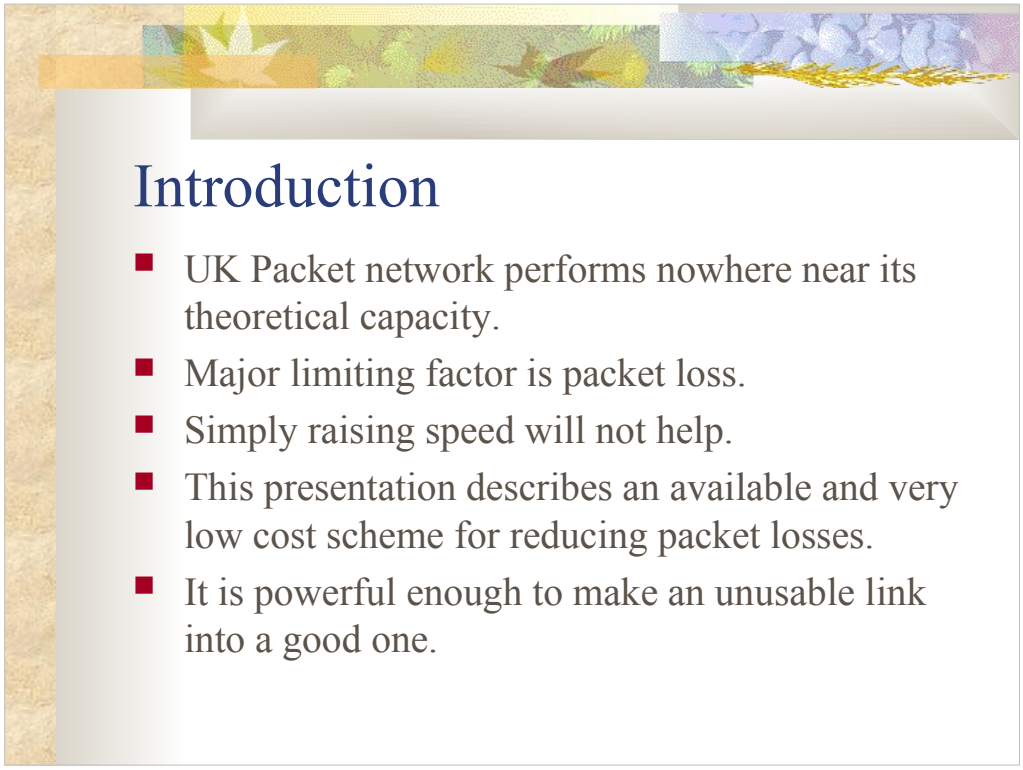
Forward Error Correction for Amateur Packet Radio

- A Software Solution

Paula Dowie G8PZT

5th April 2003



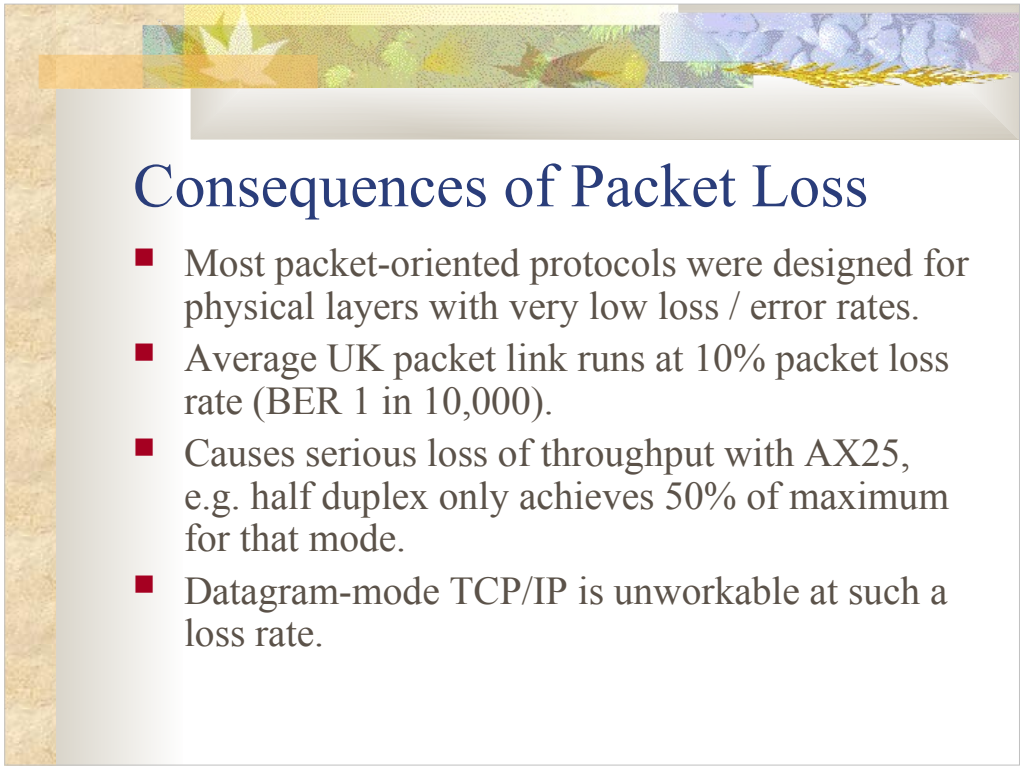
When you consider how many links there are in the UK Packet Radio network, and how little data there is to be moved, you would think that there would be plenty of spare capacity.

However, the network as a whole does not perform well, and it would appear that the full potential of the network is not actually being realised.

In my view, the major limiting factor, regardless of link speed, is packet loss. Of all the packets launched into the ether, too many are simply lost, wasting a precious resource. The performance degradation caused by these losses is many times greater than the rate of loss.

This presentation outlines the causes and effects of packet loss, then describes a very easy and low cost scheme for reducing packet losses, simply by using better software with the existing hardware.

The system is so powerful that it can turn an unusable link into a good one.



Although all packet-oriented protocols are designed to cope with lost or corrupt packets, in most cases they were designed for physical layers with very low error rates and high bandwidth, such as wire or microwave links.

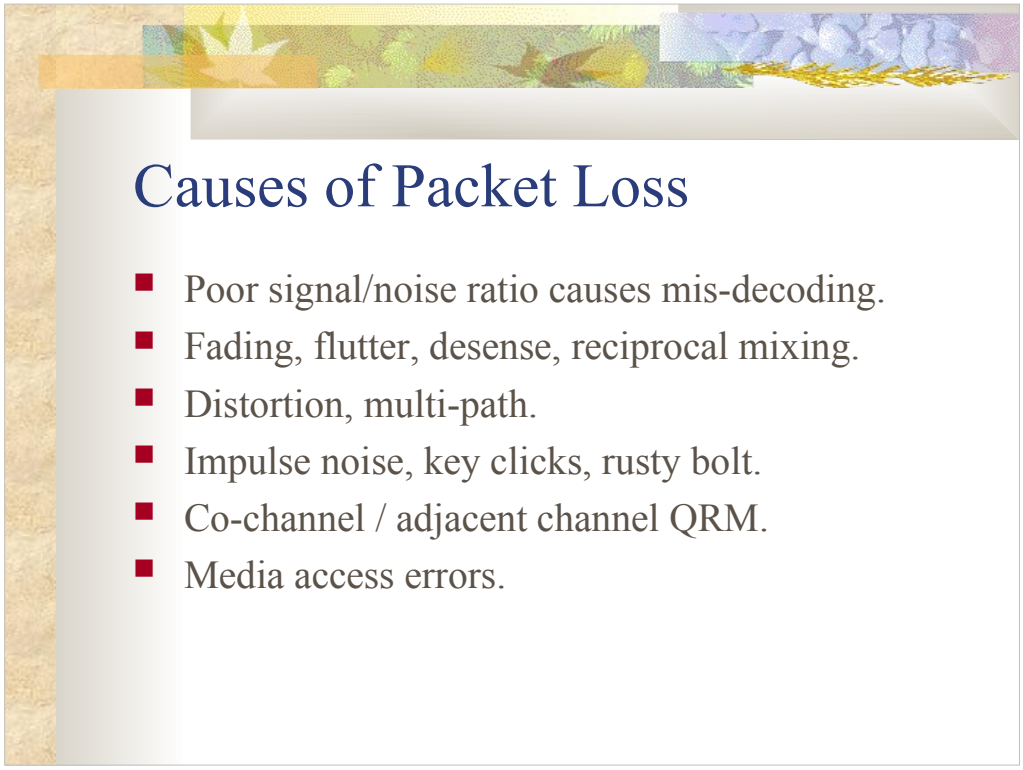
The designers assumption therefore was that a lost or corrupt packet was relatively rare, and there would be plenty of bandwidth to accommodate any re-sends.

However, in the amateur packet radio network, error rates are many orders of magnitude greater, simply because we have to work with weaker signals, inferior equipment and poor sites. What's more, we have to do it using a very low bandwidth.

I estimate that the average link in the UK network typically runs with a 10 percent packet loss rate. Before you bite my head off, I appreciate that some are much better than that, but a good many are a lot worse.

That sort of loss rate would typically degrade a 1200 baud half duplex AX25 link to half its potential throughput, and more importantly to real-time users, it doubles the average round trip time.

Whilst AX25 will still work at such loss rates, TCP/IP does not cope so well. Datagram-mode TCP/IP will be unworkable, and clearly, as the future is TCP/IP, we must get the loss rates down.



There are many causes of packet loss, some of which can be minimised by good engineering, and some which may be beyond our control.

Packets are mainly lost when the signal to noise ratio is too low for the modem to correctly decode all the data bits. This causes errors which tend to be randomly distributed through the packet.

Poor signal to noise ratio may be due to a consistently weak signal, or as a result of a temporary propagation degradations such as fading and flutter, or desensitisation of the receiver by a local transmitter.

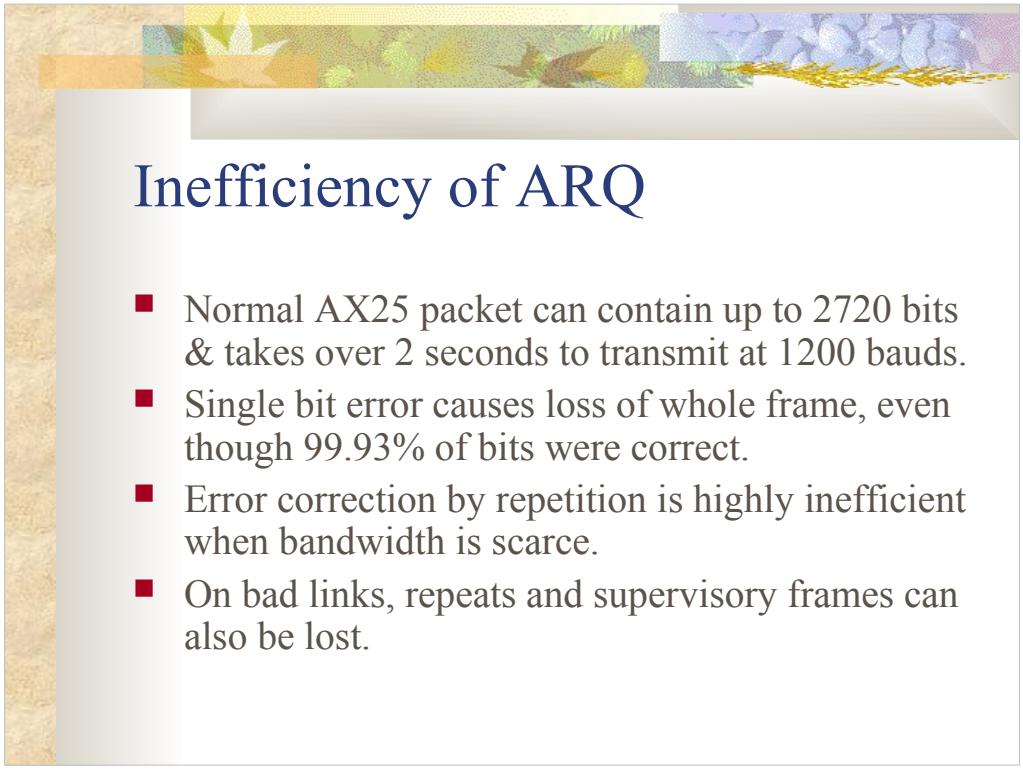
Any distortion of the signal, due to multi-path propagation, over-deviation, poor IF or AF filter responses, or distorted audio, can confuse the decoder, effectively degrading the signal to noise ratio.

Even if the steady state signal is adequate, groups of bits may be damaged by impulse noise from vehicles, household appliances and electrical machinery, or the crackles from rusty-bolt effect.

The signal / noise ratio may also be degraded by the presence of unwanted signals within the receiver pass-band, such as other stations, sidebands from adjacent channels, or inter-modulation effects.

If the antenna gain is high, it is also possible for the noise floor to be raised by solar noise at the times of day when the antenna looks into the sun.

Finally, on a half-duplex link, packets may be lost simply because both stations mistakenly transmit at the same time.



Not counting the stuffed bits, a normal AX25 packet can contain up to 2720 bits, which takes over 2 seconds to transmit at 1200 bauds.

If just ONE of those bits is mis-decoded, the CRC check fails and the frame is discarded. The 2719 correct bits, and all the time taken to transmit them is completely wasted, even though 99.93% of the packet was error free.

Moreover, it takes a significant time to detect the lost frame, and yet more time to request and receive a re-transmission. There is no guarantee that the re-transmitted frame will be error free, and if supervisory frames are lost, a lot of time can be wasted.

This process is highly inefficient when bandwidth is so limited.

Wouldn't it be nice if we could USE all the error-free bits from the packet, instead of throwing them away? Fortunately there is such a method, called Forward Error Correction…

Forward Error Correction

- "The cat sot on the mat" contains an error, but the brain automatically corrects it.
- Data contains redundant information, e.g. known words and context, which is used to guess the intended meaning.
- FEC sends extra information with data, allowing it to be reconstructed if damaged.

The sentence "The cat SOT on the mat", contains an error, yet you have no problem making sense of it, I hope!

Although the actual data consists simply of 22 ascii characters, one of which is wrong, they are grouped into standard sequences (called words) which make it easy to spot the error, and correct it by substituting the faulty character to produce a valid sequence.

If the faulty word is a near match to several alternative words, e.g. sAt, sEt, sIt, or even if the word is completely garbled, we can often guess the original with a high degree of certainty because the words themselves form sequences, called SENTENCES, which tend to obey the complex laws of grammar.

Thus "The cat SET on the mat" is rejected because it doesn't make sense, and "The cat SIT on the mat" is rejected because the tense is wrong, leaving us with "The cat SAT on the mat", which satisfies the rules of grammar, and the alliteration confirms that we have the correct word.

Sophisticated Forward Error Correction methods work in a similar way, by encoding the data using unique sequences which can be reconstructed if they get damaged. This is usually done by sending extra information along with the data, which is why it is called FORWARD error correction.

The decoder can use this extra information to reconstruct the original data if it becomes corrupt.

Some FEC Methods

- Send duplicate data & vote on errors – inefficient.
- Hamming code – matrix-based parity – limited and inefficient..
- Convolutional code - shift register based – good, but large overhead.
- Block code – e.g. Reed-Solomon – good performance, low overhead.

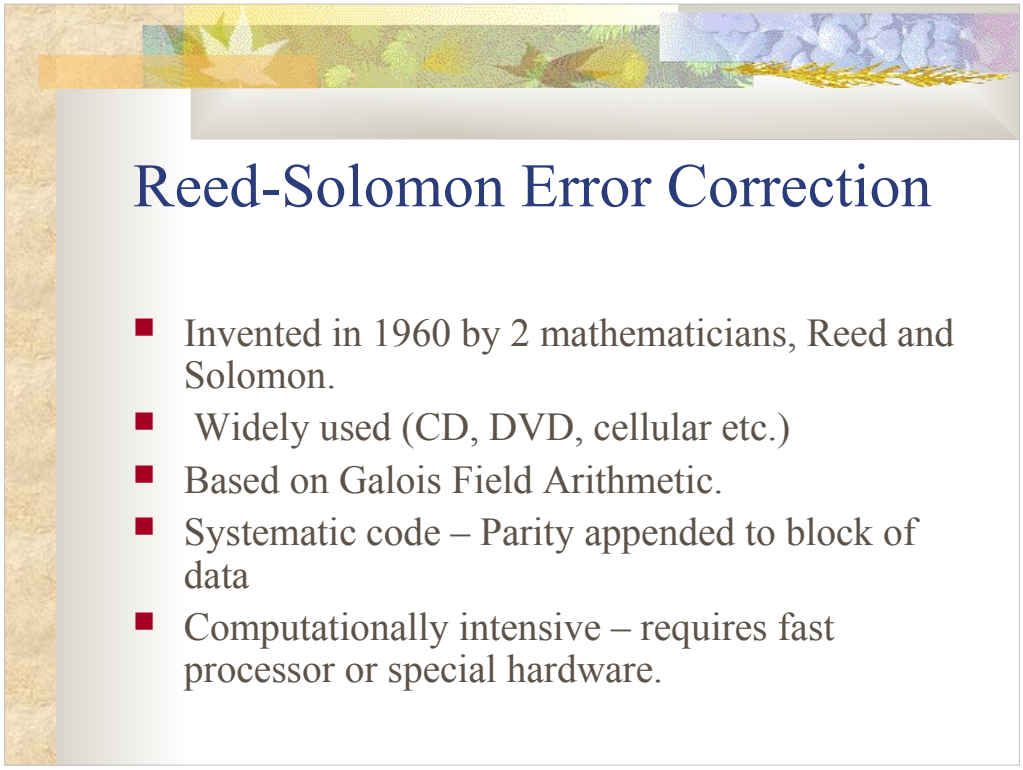
The simplest form of forward error correction is to send everything twice or even 3 times, and replies on the statistical unlikelihood of an error occurring repeatedly in the same place. This is very inefficient, and not very reliable.

Hamming Codes are generated by calculating parity on different combinations of the data, using matrices, and sending that parity with the data. The error correction capacity is useful, but small compared to the overhead.

Convolutional codes operate on serial data, a few bits at a time. They are generated by sending the data through a shift register with feedback, such that any bit entering the register has an effect on the subsequent bits, and during this process a given number of input bits gives rise to a larger number of output bits, thus adding redundancy. This is a much more powerful technique than the Hamming Code, but also has a large overhead.

Convolutional codes are well suited to dealing with signals contaminated by white noise, but not so good for dealing with burst errors.

Block Codes such as Reed-Solomon calculate a complex form of parity over a relatively large block of data, and are particularly good at correcting burst errors, whilst still performing well on Gaussian noise. The overhead is relatively small for a good error correction capacity, so I feel it is best choice for Packet Radio.



Reed Solomon codes were invented at MIT in 1960 by two mathematicians, and have since been widely used in many applications, such as storage devices and communications.

They are linear block codes based on Galois field arithmetic, and the code-blocks are generated by a special polynomial, such that all valid code-blocks are exactly divisible by the polynomial.

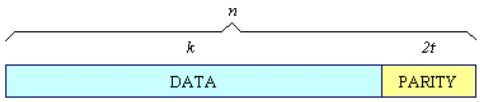
As far as I can remember, the “parity” information is derived from the roots of an equation which describes the original data.

The data itself is unchanged, and the parity information is appended to it. This is called a “systematic code”.

Reed-Solomon error correction is computationally intensive, especially the decoding, and requires plenty of processor power, or special encoder / decoder hardware.

There are now chips designed for this purpose, but fortunately modern PC’s are fast enough to do it, and I have chosen to implement the error correction in software.

Reed-Solomon Codeword



- A Reed-Solomon codeword consists of a block of data “symbols”, with parity symbols appended.
- Maximum total block size for 8 bit symbols is 255 bytes ((2**8)-1).
- Error Correction Capacity (ECC) is the no. of parity symbols divided by 2.
- Expressed as RS(n,k), e.g. RS(255,239)

A Reed-Solomon codeword consists of a block of data “symbols”, with parity symbols appended.

Symbols are comprised of a number of data bits, which may or may not equal the data byte size. For simplicity I have used 8 bit symbols.

The maximum total block size for a given symbol size is (2**x)-1, where x is the number of bits per symbol. For 8 bit symbols this is 255 bytes ((2**8)-1).

The Error Correction Capacity (ECC) of the code is the no. of parity symbols divided by 2, and I will refer to this later.

The characteristics of a Reed Solomon code block are usually expressed in the form RS(n,k), where n is the total block size and k is the number of data symbols, e.g. RS(255,239).

The first figure implicitly tells us that the symbol size is 8 bits, and the number of parity symbols can be deduced by subtracting the second figure from the first.

Symbol Errors

- One symbol error occurs when 1 or more bits in a symbol are wrong.
- RS(255,239) can correct up to 8 symbol errors,
- In worst case this could be 1 bit in each of 8 symbols (8 bit errors), or in best case 8 bits in each of 8 symbols (64 bit errors).
- If the 8 symbols are consecutive, it will correct a run of 64 faulty bits, I.e. a burst error.

One symbol error occurs when 1 or more bits in a symbol are wrong.

As mentioned previously Reed-Solomon can correct a number of symbol errors equal to half the number of parity symbols, so a code such as RS(255,239), which uses 16 parity symbols can correct up to 8 symbol errors, anywhere in the codeblock

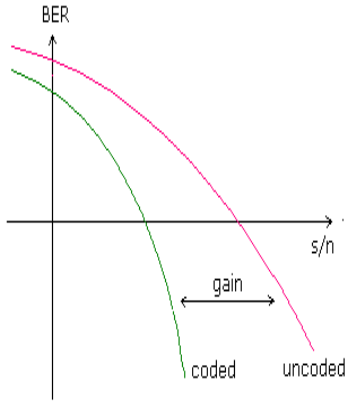
In the worst case this could be 1 bit in each of 8 separate symbols, and the code would thus correct 8 single bit errors.

In best case it could correct all 8 bits in each of 8 symbols, a total of 64 bit errors.

The 8 symbols could be consecutive, in which case it will correct a run of 64 faulty bits, which could typically arise from a click.

Coding Gain

- Probability of an error in decoded data block is lower with FEC than without it.
- Equivalent to an increase in signal to noise ratio.
- Gain is usually expressed in decibels.
- Improved performance for given SNR, or same performance for lower SNR.



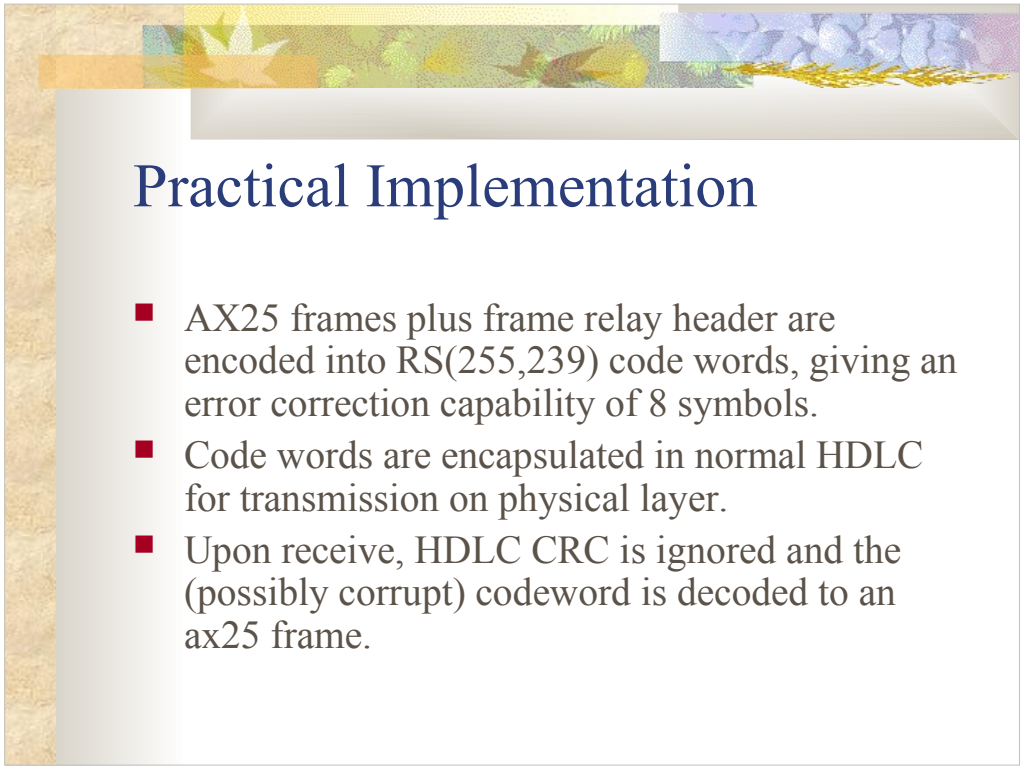
The probability of an error in a decoded data block is lower if FEC is used, than if it is not used.

The improvement in error rate is equivalent to increasing the signal to noise ratio. **Coding gain** is the difference in signal-to-noise ratio between a coded channel and an uncoded channel, and is usually expressed in decibels.

This gain can either be used to get an improved performance from a channel with a given signal to noise ratio, or to obtain a given performance from a channel with lower signal to noise ratio.

For example on a good link it allows the use of lower transmit powers, smaller aerials, longer packets, or greater fade and noise immunity.

But in amateur service it is more likely to be used to improve a poor link.



The scheme I have implemented in Xrouter is basically as follows:

Firstly, ordinary AX25 frames are given a 4 byte frame relay header, then encoded into RS(255,239) code words, using 16 bytes of parity to give an error correction capability of 8 symbols.

This was deemed to be the optimum compromise between error correction and overhead, given the next bit…

The code words are then encapsulated in normal HDLC for transmission on the physical layer.

Upon receive, the HDLC Cyclic Redundancy Check is ignored and the possibly corrupt codeword is decoded back to an ax25 frame, provided the number of errors does not exceed the error correction capacity of the code.

Short Packets

- Reed-Solomon code words are fixed size, in this case 255 bytes.
- AX25 packets can be a lot shorter.
- Data is padded with zeros before parity is computed.
- Only data and parity are transmitted.
- Received data padded with zeros before parity is checked.

Reed-Solomon code words are based on fixed block sizes, in this case I am using 255 symbols of 8 bits each.

But AX25 packets can be a lot shorter than this, down to a minimum of 15 bytes in fact.

It would clearly be inefficient to transmit a 255 byte code block for a 15 byte packet.

Fortunately Reed-Solomon coding allows us to transmit short packets.

The data is first padded with zeros to form a maximal-sized data block, then the parity over this block is computed in the normal way.

Instead of transmitting the whole code word, we only need to transmit the original data plus the parity, not the padding.

Upon receive, the short data is padded with zeros to form a maximal data block before decoding normally.

Long Packets

- Maximum AX25 packet is 336 bytes.
- RS(255,239) code word can only accommodate 239 data bytes.
- 9 bit symbols require data conversion & packet size still limited to 555 bytes.
- Solution: Packets greater than 239 bytes encoded into multiple equal-sized blocks – infinitely extendable.

Conversely, the maximum AX25 packet size is 336 bytes, but the Reed Solomon code word I've chosen to use can only accommodate 239 data bytes.

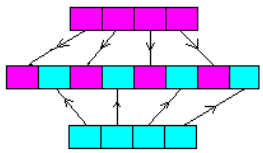
One solution would be to use 9 bit symbols, for which the code word size would be 511 symbols.

This would accommodate 555 bytes of data, but adds extra complexity because the data must be converted back and forth between 8 and 9 bit symbols.

Although this would accommodate a standard AX25 packet, it still imposes a limit on packet size.

The solution I've chosen is to encode large packets into two or more code words, because that is infinitely extendable.

Interleaving



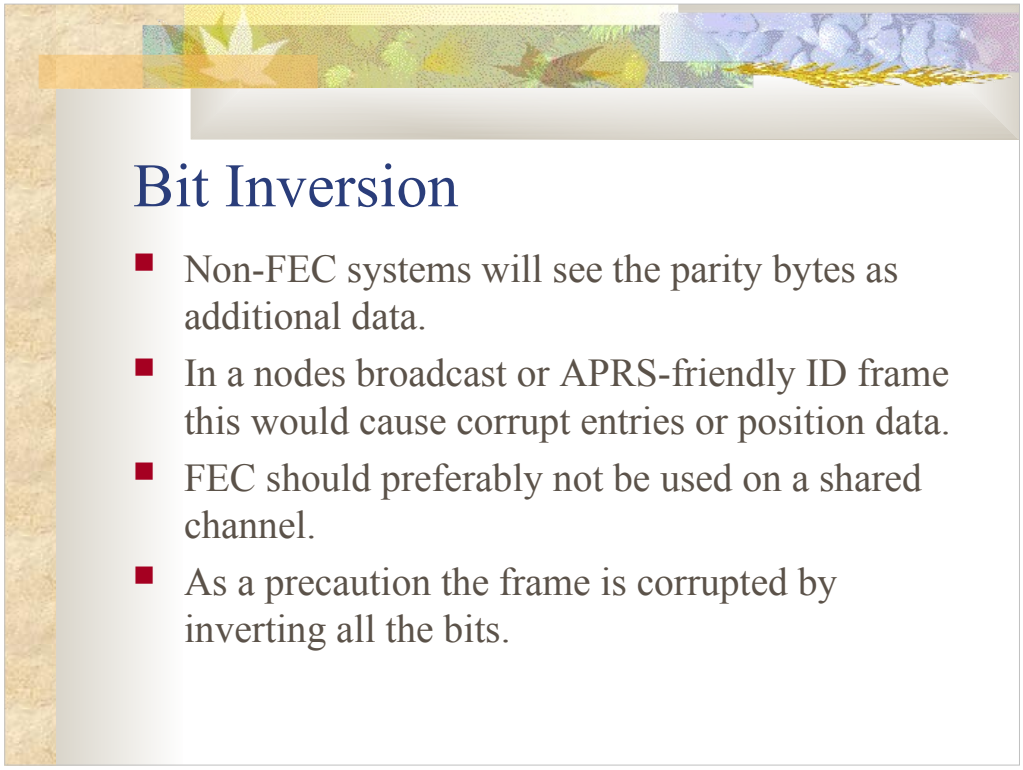
- Only used when a packet is encoded into multiple blocks.
- Bytes from each code block are sent alternately.
- Spreads burst errors across code blocks, so they're less likely to exceed ECC of any one block.

When a large packet is encoded into more than one code word, they could simply be sent sequentially, but I have chosen to interleave them.

Interleaving basically sends bytes from each codeword alternately, and the effect of this is to spread burst errors across the code words, so they're less likely to exceed the error correction capability of a single block.

The diagram shows two codewords, in cyan and purple, and how they are interleaved to form the transmitted packet. The process is reversed on receive.

If you imagine a burst error affecting 4 consecutive bytes, you will see that half of them will be from the purple codeword, and half from the cyan codeword,



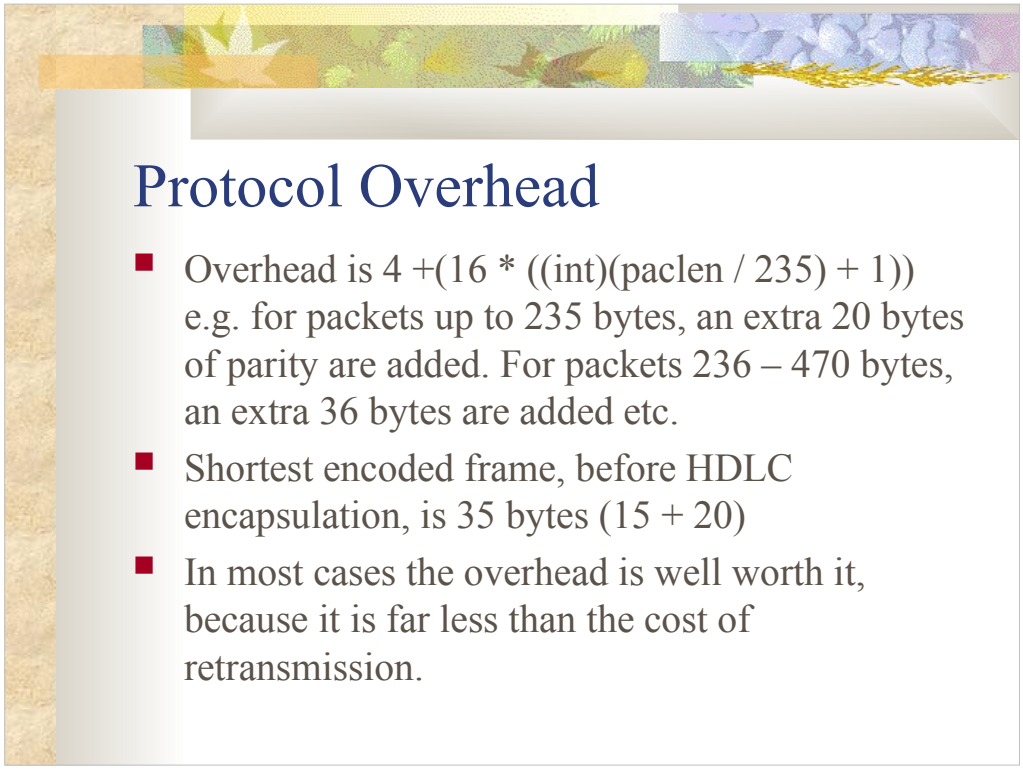
Since a valid CRC is generated when the blocks are transmitted, they will be received normally on vanilla AX25 systems, but they will see the parity bytes as additional data.

Most AX25 implementations will ignore the extra bytes on supervisory frames, but it is possible that some might crash.

Furthermore, on unconnected mode packets such as nodes broadcasts or ID frames, the parity bytes can be interpreted as node entries or APRS position data.

For these reasons, Forward Error Correction should preferably not be used on channels shared with regular AX25 systems.

But in case this is unavoidable, I have taken the precaution of inverting all the bits in the frame, thus rendering it invalid as far as vanilla systems are concerned, because the callsigns and control byte should then fail the various validity checks, assuming the software is good enough to have such things!



Each AX25 packet is given a 4 byte frame relay header, and the Reed Solomon coding adds a further 16 bytes for each code word used.

This amounts to 20 bytes of overhead for packets up to 235 bytes, which is the majority of them, and 36 bytes of overhead for packets over 235 bytes.

Thus if the shortest AX25 frame is 15 bytes, the shortest encoded frame is 35 bytes.

This may seem like a lot, but it represents less than 10 percent of a large packet, and is well worth the extra bytes because the cost of re-transmission is far greater.

HDLC Framing

HDLC Framing

- HDLC chosen for compatibility with existing hardware (SIO chips, YAM, KISS TNC's etc.)
- If flags are damaged, the frame is lost.
- Probability of mis-framing is much lower than probability of error in payload.
- CRC not needed, but SIO/SCC chips force its use.

HDLC is not the best method for framing a forward-error-correcting packet, but it was chosen because it is understood by lots of existing hardware.

My view was that any system which relies on special hardware is doomed to a poor uptake. I would rather have widespread use of a good, but less than perfect, system, than a state of the art system which no-one uses.

The main problem with HDLC is that if either of the two flags which delineate the frame are misdecoded, the frame will not be recognised as such. Furthermore, if any data byte is misinterpreted as a flag, the frame may be prematurely truncated.

Fortunately, the probability of such effects is much lower than the probability of an error in the payload, so forward error correction is still a viable prospect.

The two CRC bytes are not needed, and it would have been nice to do away with them, but the most common SIO and SCC chips do not allow them to be disabled., so they are simply ignored.

Requirements (each end)

- Xrouter packet router software
- Plus an SCC Card
- Or YAM Modem
- Or TNC with modified BPQKISS firmware
- But some TNC's using 'RUH modems may require a squelching mod unless RS232 data rate is high.
- Could use Baycom modem if their driver program was modified

My Forward Error Correction scheme uses Xrouter, and some fairly standard hardware. Both ends of the link must run Xrouter with FEC switched on. This is a simple configuration command, and it can be switched on and off on the fly.

It will run on SCC cards or YAM modems, and may also be used with TNC's, providing they are using modified BPQKISS firmware.

Some TNC's using G3RUH decoders may require modification to prevent the RS232 line being overloaded by garbage frames.

This is because the RUH decoder generates copious quantities of spurious HDLC flags whilst it is idling. Without the CRC check, these are passed to the serial line, and if the line speed isn't fast enough, the TNC will fill up and reboot itself.

To prevent this, the serial line should be run at very high speed, or ideally the circuit could be modified to gate off the input to the SIO chip when the decoder is not in a synchronised state.

A Baycom modem could be used providing someone modifies the L2 driver program to disable the CRC check. One day I might get around to doing it myself.

Results

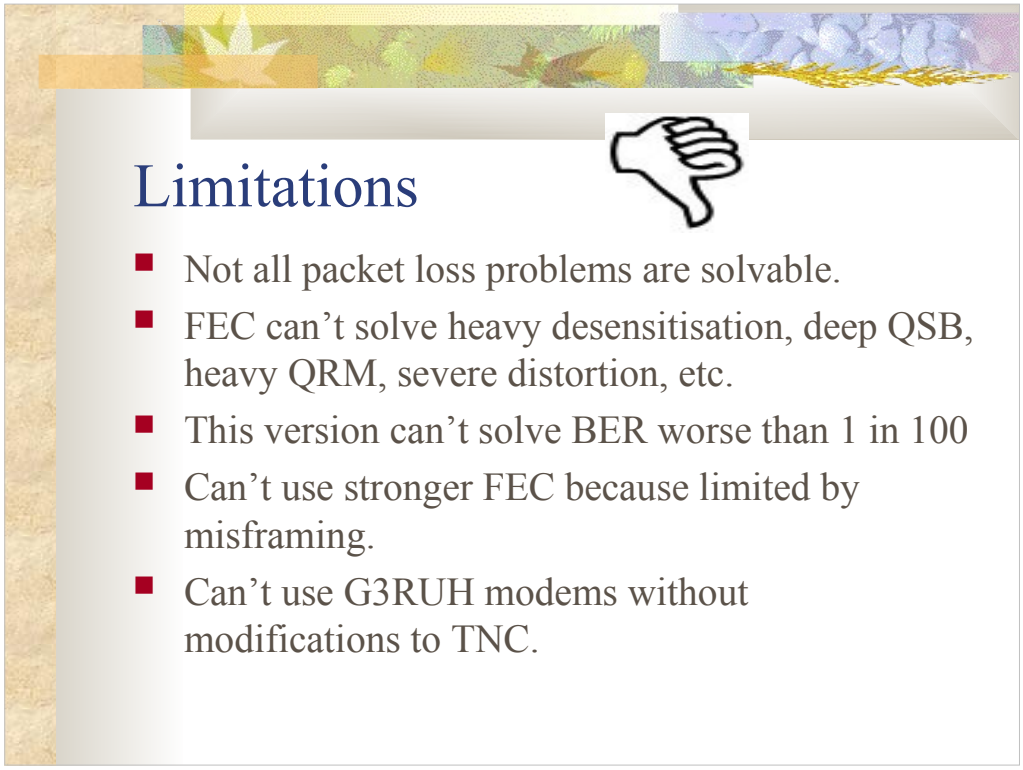
- System tried over several poor RF links, and on a link simulator.
- In every case the reductions in packet loss and improvements in link throughput were dramatic.
- E.g. able to run datagram-mode TCP/IP over a link which, without FEC, couldn't maintain an AX25 connection.

The system has been tried over several RF links, using combinations of SCC cards, YAM modems and BPQKISS TNC's.

It has also been tried on a link simulator program designed to produce predictable bit error rates.

In every case, the reductions in packet loss, and the consequent improvements in link throughput have been dramatic.

For example, in one of the harshest tests, we were able to run a demonstration of internet via packet, using datagram-mode TCP/IP, over a link which couldn't even sustain an AX25 connection without the Forward Error Correction.



Forward Error Correction is not the panacea for all link problems. It must be understood that not all packet loss problems are solvable by software alone.

For example FEC can't solve heavy desensitisation, deep QSB, heavy QRM, severe distortion, etc. Whilst it can re-create up to 8 lost bytes, it can't create a whole packet from thin air. These are heavy engineering problems.

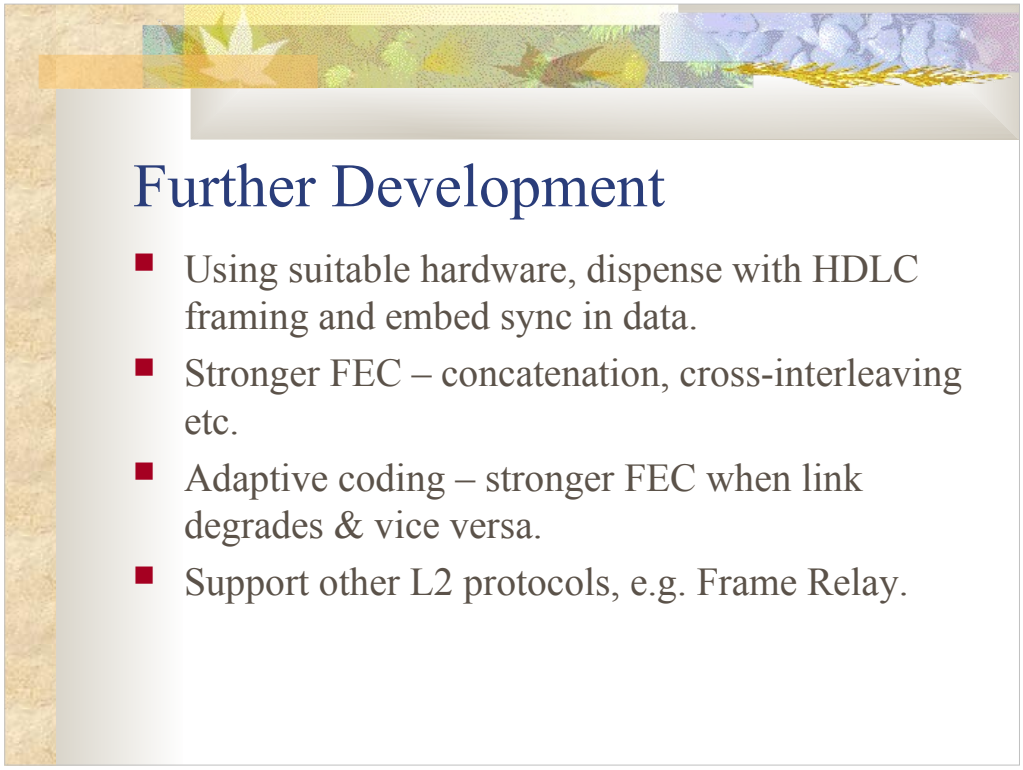
Because this version was required to use HDLC framing, the error correction gives little benefit at bit error rates which are worse than approximately 1 in 100, because that's the point at which packets are more likely to be lost by HDLC framing errors than by correctable errors in the frame contents.

However, putting this into perspective, that Bit Error Rate is 100 times greater than the average packet link suffers.

It is pointless using stronger error correction techniques on HDLC, as they would all suffer the same upper limit due to the misframing problem.

It must also be understood that this system can never achieve an absolute zero loss rate, because of the HDLC problem, but on a reasonably well engineered link, 1 lost frame in 10 million is easily achievable.

Lastly, as already mentioned, you can't easily use this system with G3RUH modems unless you modify the TNC.



I must stress that this implementation of Forward Error Correction is merely a first version, a quick and easy fix that anyone can use. There is still plenty more I plan to do with it.

For example, the next version will dispense with the HDLC framing and instead will use sync markers embedded in the data to delineate the frame.

This should provide more robust framing, allowing the use of even stronger FEC techniques, such as cross-interleaving, or the concatenation of Reed-Solomon and convolutional codes.

But it will require suitable hardware which is not based around an SIO or SCC chip, for example a reprogrammed YAM modem, or something completely new. There are plenty of new chips to be explored.

Since the overhead of very strong FEC is not always justified, I plan to make the coding adapt to the link conditions, so that stronger FEC is used when the link degrades and vice versa.

Although I'm currently using them to carry AX25, Reed-Solomon code words don't care what the payload is. Having created a new packet structure which, from the outset, includes an extensible payload identifier and link control flags, I propose to experiment with alternative layer 2 protocols, such as frame relay.

The End

Thank you for listening