AX.25 Throughput: Why is 9600 bps Packet Radio only twice as fast as 1200?

(and what can we do to fix it?)

WB2OSZ, 1/22/2019

If you find this document somewhere else, the most recent version should be available at:

https://github.com/wb2osz/direwolf/tree/dev/doc

Amateur Packet Radio is very useful for emergency communications but it is falling behind with increasing demands. A long time ago, someone might have been impressed by the ability to pass a 25 word message through multiple relay stations but now expectations are a lot higher. It is reasonable to expect that lengthy text can be sent along with attachments such as a photograph for damage assessment or a spreadsheet to order supplies for a shelter.

Some day we might all have gigabit speed ham radio mesh networks but for now most of us need to settle for squeezing what we can out of repurposed radios designed for voice. The most obvious speedup would be to move from 1200 bits per second (bps) to 9600. The 9600 bps standard has been around since the late 1980's. Radios with a built-in TNC (TM-D700 family, etc.) all have 9600 bps capability. Many traditional specialized hardware, and newer software, TNCs have 9600 capability. It's a mature, widely available standard but it is not being used that much.

Simply switching to a higher data rate will result in great disappointment. You might naïvely expect it to be 8 times faster but it can turn out to be only twice as fast.

In this document we look at why a large increase in data bit rate produces a much smaller increase in throughput. We will explore techniques that can be used to make large improvements.

Let's start with the basics. After that we will look at experimental results of changing the various parameters.

1 The AX.25 Frame

Data is transmitted in “frames” (sometimes called packets) like this:

Flag 01111110	Destination Address	Source Address	Optional Digipeaters	Control & Protocol	Information	Frame Check (CRC)	Flag 01111110
1 byte	7 bytes	7 bytes	0 – 56 bytes	1 – 3 bytes	0 – 256 (or more) bytes	2 bytes	1 byte

- It starts with a unique pattern, called “flag,” which can’t appear inside the frame.
- Addresses indicate the intended recipient and sender. Usually ham radio callsigns.
- Optional digital repeaters (“digipeaters”) can be used to relay the frames longer distances.
- Control bytes determine the function and how to interpret the Information part.
- The Information part is where the useful data is carried.
- The Frame Check Sequence is used for error checking. Frames with errors are discarded.
- Finally, it ends with another “flag” pattern.

Complete details can be found in the AX.25 protocol specifications:

- Version 2.0, October 1984
- Version 2.2, July 1998 ← more than 20 years ago.

The newer version has several improvements that make bulk transfer of data more efficient. Products designed in the previous Century, and not updated, would not have these improvements.

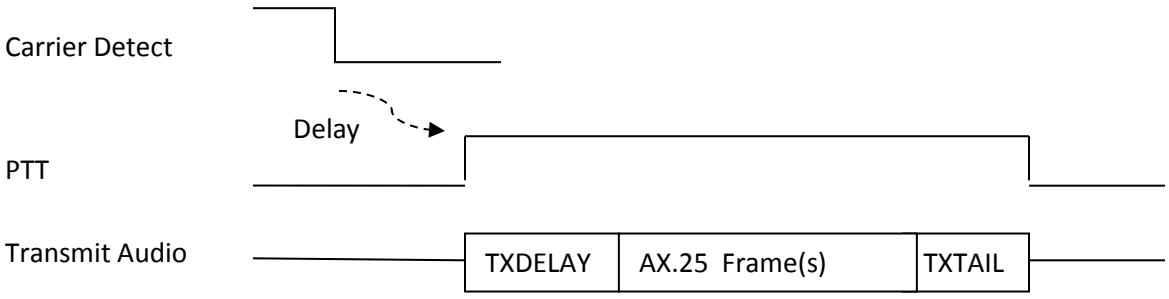
2 Transmit Timing

Transmit timing is determined by several parameters. Some typical values are:

SLOTTIME 10	x 10 mSec per unit = 100 mSec.
PERSIST 63	probability for transmitting after each slottime.
TXDELAY 30	x 10 mSec per unit = 300 mSec.
TXTAIL 5	x 10 mSec per unit = 50 mSec.

The factor of 10 in there can cause confusion. Why not simply list the number of milliseconds? The earliest implementations used 8 bit microprocessors where memory and processing power were quite limited. Numeric values were scaled so they could fit into a single byte. That's why you see many parameters can be in the range of 0 to 255. Sometimes scaling is applied to fit more convenient numbers into that range.

So, it is just tradition of following the way it has always been, since the beginning, for the most fundamental configuration commands.



When a frame is ready for transmission, we first have to wait until the channel is clear. The technical term for this is Carrier Sense Multiple Access (CSMA). In plain English, if you want to say something, wait until no one else is speaking.

- For digipeated frames the transmission can begin immediately after the channel is clear. AX.25 "connected" mode also has some situations where "expedited" frames go out immediately rather than waiting a random time.

- Most of the time, SLOTTIME and PERSIST are used to generate a random delay. This is to minimize the chances of two different stations starting to transmit at the same time. The process is:

    (a) Wait for SLOTTIME.

(b) If a random number, in the range of 0 to 255, is less than or equal to PERSIST, start to transmit. Otherwise go back to step (a).

For the typical default values, we have delays with the following probabilities:

Delay, mSec	**Probability**	
100	.25	= 25%
200	.75 * .25	= 19%
300	.75 * .75 * .25	= 14%
400	.75 * .75 * .75 * .25	= 11%
500	.75 * .75 * .75 * .75 * .25	= 8%
600	.75 * .75 * .75 * .75 * .75 * .25	= 6%
700	.75 * .75 * .75 * .75 * .75 * .75 * .25	= 4%
etc.	...	

(c) If a signal is detected during any of the steps above, we go back to the top and start over.

(d) The Push to Talk (PTT) control line is asserted.

The frames can't be sent immediately because the transmitter takes a little while to stabilize after being activated. For example, the same phase locked loop might be used for both transmit and the receiver local oscillator. You wouldn't want to send out a signal until this settles down.

There can also be delays at the receiving end. If squelch is used, there will be a delay before received audio comes out. To make things worse, the audio amplifier might be powered down when not needed. When the power is restored there can be a huge transient that takes time to settle down. Of course, there can still be the matter of the frequency synthesizer settling down.

The HDLC "flag" pattern (01111110) is repeated for a time period of TXDELAY. In my testing, I found 200 mS was too short for a typical 2 meter transmitter / receiver combination. 300 mSec (TXDELAY 30) is generally a good starting point.

When sending audio out through a "soundcard" there is latency between sending an audio waveform to the output device and when the sound comes out. We can't be sure precisely when the queued up sound has been completed so we need to keep the PTT on a little longer just to be safe. The HDLC "frame" pattern is also sent during this TXTAIL time to keep the channel busy.

Don't get too stingy here. The same setting that works with a particular receiver and receiving TNC might not work with a different receiver or TNC.

If you think you can get significant gains by trimming this to the absolute minimum workable value, you are barking up the wrong tree.

3 Channel Efficiency

Suppose you wanted to transmit a message of 80 bytes. How long should that take? You might figure 80 bytes x 8 would be 640 bits. At 1200 bits per second, it should take 640 / 1200 = 0.53 second. The "Information" part would take that long but we have a lot of overhead, about 20 byes per frame. There are also the random wait and TXDELAY times.

Roughly,

- Average about 0.25 second for the random wait.
- About 0.3 second for TXDELAY.
- 100 byte frame (* 8 bit per byte / 1200 bits per second), 0.67 second.
- TXTAIL 0.05 TXTAIL.

Only 0.53 seconds, out of a total of 1.27 were for useful data. About 42% of the time. The rest is in wasted overhead.

What happens if we increase the speed to 9600 bits per second?

Roughly,

- Average about 0.25 second for the random wait.
- About 0.3 second for TXDELAY.
- 100 byte frame (* 8 bit per byte / 9600 bits per second), 0.083 second.
- TXTAIL 0.05 TXTAIL.

Only 0.083 seconds, out of a total of 0.683 were for useful data. About 12% of the time. The rest is in wasted overhead.

The total time was roughly cut in half. Another way to look at it is that about twice as many transmissions can be sent in the same amount of time. Only 2 times as fast. Not 8 like one might naively hope for.

The reason is that the fixed overhead does not decrease along with the faster bit rate.

It can be even worse with connected mode packet. It is not just a one way transfer. The receiving end sends back acknowledgements that the frames were received correctly. This allows automatic retransmission if something gets lost.

Let's have a quick review of the difference between APRS and connected mode packet radio.

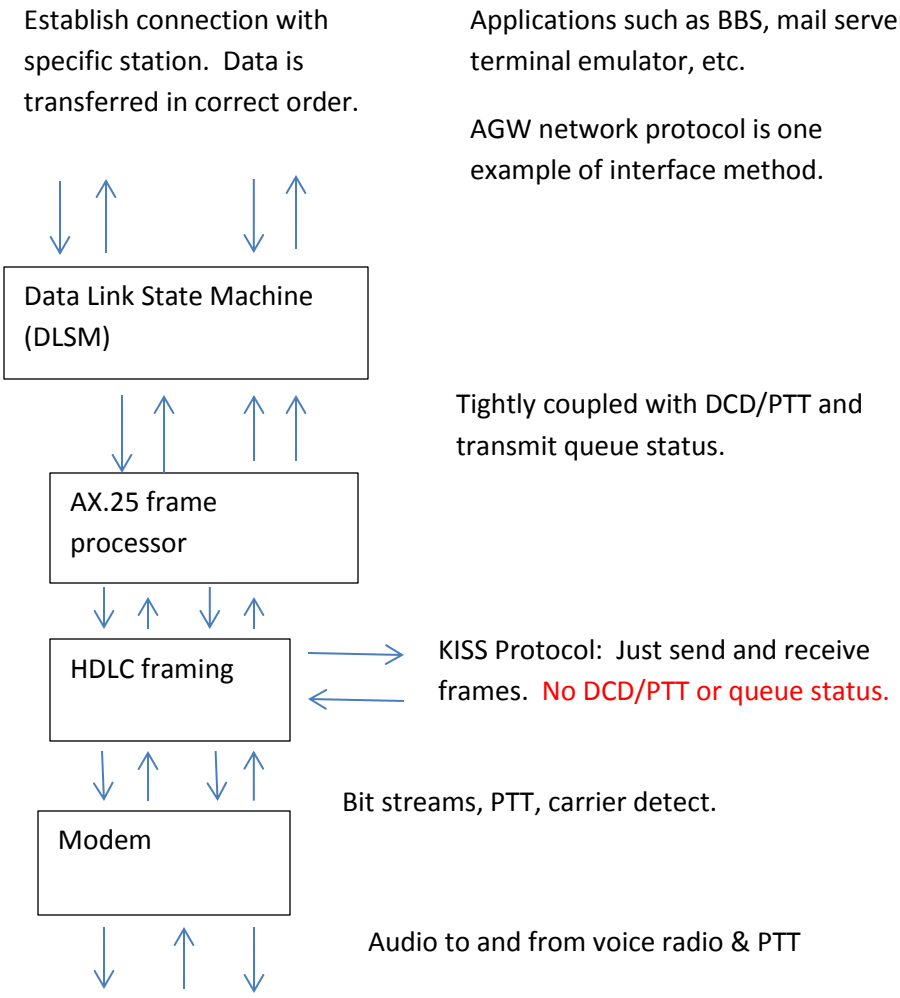
4 APRS vs. Connected mode Packet

The AX.25 protocol is used mainly in two ways:

APRS	Connected mode
One to many.	One to one.
Usually no particular destination.	Establish a link with a specific station.
Send and forget independent (unnumbered) information frames.	The link protocol assigns sequence numbers to the information frames.
Don't know if frames were received by anyone.	The receiving end replies with acknowledgements that the frames were received. Lost frames are resent automatically and delivered in the proper order.

The combination of a modem, and a processor for the AX.25 protocol, is called a Terminal Node Controller (TNC).

A TNC has different parts that can be visualized like this:



At the bottom we have the modem which sends digital bits through voice channels. The AX.25 protocol specification says absolutely nothing about the modems to be used. In practice we mostly use 1200 baud AFSK (based on Bell 202). For 9600 bps, it's really a two level baseband signal with pseudo random scrambling to reduce the DC component. This is often referred to as G3RUH for one of the early hardware designs.

Above the modem we have the HDLC framing with:

- Unique start pattern 01111110.
- Variable number of bits.
- Frame Check Sequence / CRC for error checking.
- Same unique pattern again to indicate end.

At this level, the "KISS" protocol interface is often available. The rest of the AX.25 protocol is handled by a separate computer application, not in the TNC.

It is very simple.

- For transmit, the application (or higher level protocol) says, "Transmit this frame when you can." The sender has no idea when the frame is transmitted, or if it got discarded due to lack of memory.

- In the receive direction, incoming frames are reported. There is no information about the carrier detect so you don't know if another is in progress or if the incoming transmission has ended. You don't know if the channel is tied up by other station so an expected reply will be delayed.

Above that, the AX.25 frame processor defines the format of the HDLC payload:

- Addresses.
- Control.
- Information part.

APRS and digipeaters can be built on top of that. They don't deal with persistent connections and the state information that is required.

Above that, we have the "Data Link State Machine" (DLSM) for connected mode.

Ideally, this will have a tight coupling with the lower level. It needs to know about the carrier detect, PTT, and transmit queue status to make the best choices. If we try to use the KISS interface here, kludges, compromises, and performance sacrifices need to be made.

Finally, at the top, we have a different type of application interface. Typical usage at this layer would go like:

- Tell the TNC to connect to some specific station such as a Bulletin Board System (BBS) or Mail server.
- Send some commands and data.
- Get some data back.
- Disconnect to end the session when done.

The Data Link layer takes responsibility for delivering the data to the other station. Anything that gets lost is resent and put back into the correct order before it is delivered.

Further Reading:

Power Point presentation explaining: Brief Packet Radio History, Hardware vs. Software TNC, KISS TNC, etc. etc.

5 PACLEN

AX.25 packet radio TNCs allow you to set various parameters to make appropriate tradeoffs.

The PACLEN parameter, called “N1” in the protocol specification, is the maximum length of the Information field of the frame.

The earlier v2.0 protocol specification had a maximum information part of 256 bytes. This was back when most home computers used 8 bit processors and couldn’t address more than 64k bytes.

In the v2.2 specification, this limit was lifted and the two stations can negotiate the maximum size. For example, the originating station might ask for a 4K byte maximum size and the other station would reply it could only handle a maximum of 2K.

Each frame has a fixed size overhead regardless of how much useful data it is carrying. It is also more efficient, for large data transfers, because the overhead becomes a smaller percentage.

Information bytes	Overhead bytes	Information part as % of full frame
64	20	64 / (64 + 20) = 76 %
128	20	128 / (128 + 20) = 86 %
256	20	256 / (256 + 20) = 93 %
1024	20	1024 / (1024 + 20) = 98%

Larger PACLEN is more efficient.

There is a tradeoff here. **A longer frame is less likely to get through under unreliable conditions.** On noisy HF SSB, you would be better off having a larger number of smaller frames because a noise burst would wipe out a smaller portion and less would need to be retransmitted.

On VHF FM, it’s not as much of a consideration. The natural noise level is lower and you can’t hear people from very far away. You are most likely to have full quieting FM signal, or something too weak to be decoded at all, with little gray area in between.

That is what we are talking about here. Transitioning from 1200 to 9600 bps on VHF/UHF. HF is a different game.

6 MAXFRAME

If Station A had 4 information frames for Station B, the communication might go like this.

Station A	Transmit Direction	Transmit Direction	Station B
Information frame 0	→		
		←	Got it; Ready for 1
Information frame 1	→		
		←	Got it; Ready for 2
Information frame 2	→		
		←	Got it; Ready for 3
Information frame 3	→		
		←	Got it; Ready for 4

This is very inefficient. Half the transmissions are an acknowledgement (ACK) that a single Information frame was received.

The MAXFRAME parameter, commonly known as the Window size, is the maximum number of Information frames that can be sent before waiting for acknowledgment. The example above has a MAXFRAME value of 1.

For a value of 2, it would go like this:

Station A	Transmit Direction	Transmit Direction	Station B
Information frame 0	→		
Information frame 1	→		
		←	Got it; Ready for 2
Information frame 2	→		
Information frame 3	→		
		←	Got it; Ready for 4

Now a single ACK can be used for two frames received correctly. Only 1/3, rather 1/2, of the traffic is acknowledgments. Two information frames can be sent adjacent to each other so the random wait and TXDELAY in between can be eliminated. It is a win-win situation.

Why stop at 2? With MAXFRAME = 4, four frames can be sent in a single transmission and the other station can send a single acknowledgement for all of them.

Station A	Transmit Direction	Transmit Direction	Station B
Information frame 0	→		
Information frame 1	→		
Information frame 2	→		
Information frame 3	→		
		←	Got it; Ready for 4

We will discuss a tradeoff (and a solution) later, but the primary lesson here is that larger is more efficient. In a later section, we will cover what happens if something is not received properly.

The AX.25 Protocol Specification refers to the window size as "k." For AX.25 v2.0, the maximum is 7 due to the 3 bit sequence number. The default is usually 4.

7 EMAXFRAME

Version 2.2 of the protocol adds "Extended" sequence numbers with 7 bits. Up to 63 frames can now be sent before waiting for an ACK. You would expect 127 for 7 bits but only half the range is usable when we introduce "Selective Reject" later. The default is 32. In a later section we will look at the impact of increasing this value.

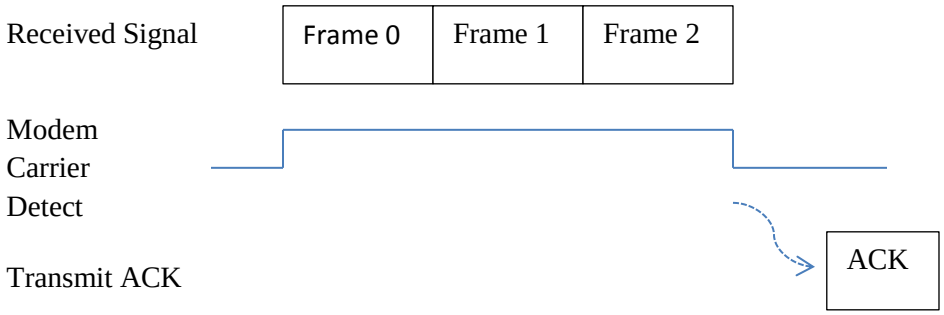
window size ("k")	v 2.0 MAXFRAME	v 2.2 EMAXFRAME
Minimum	1	1
Default	4	32
Maximum	7	63

Remember: a larger number is a win-win:

- Only a single random delay and TXDELAY for many frames sent.
- Single acknowledgement for many Information frames.

8 When to send ACK? – Smart Way

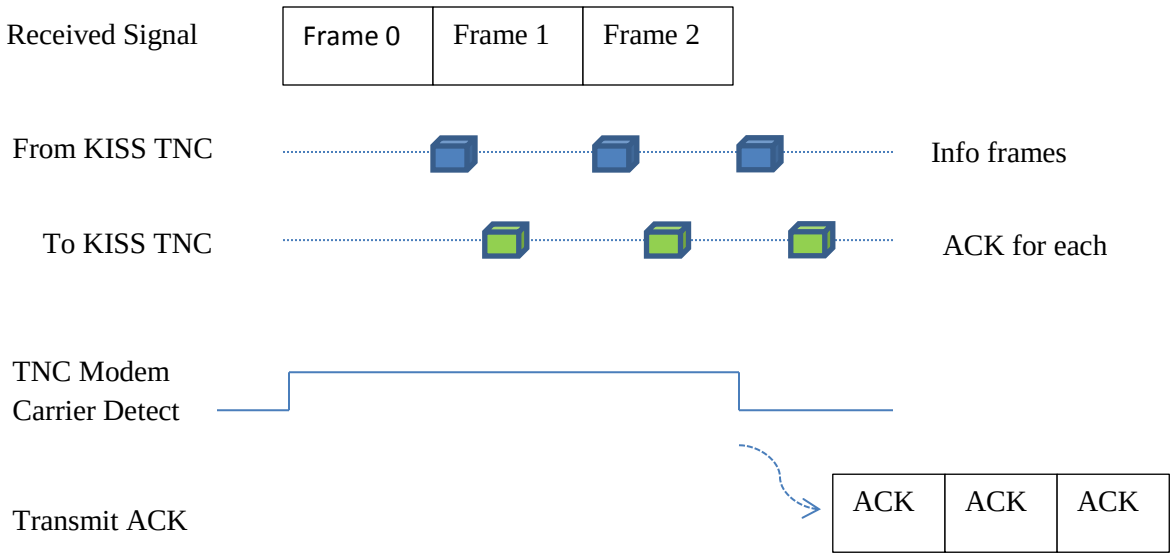
After receiving information frames, we need to inform the sender that they were received properly. The sender will then know it is OK to send new frames rather than going back and resending any that were lost.



How do you know when to send the acknowledgement? You wouldn't want to send it at the end of the first frame; there could be more in the same transmission. The obvious answer is to look at the carrier detect signal from the modem. Wait our usual random channel access time (average about ¼ second) and acknowledge receipt of them all with a single frame.

9 When to send ACK? – Stupid Way

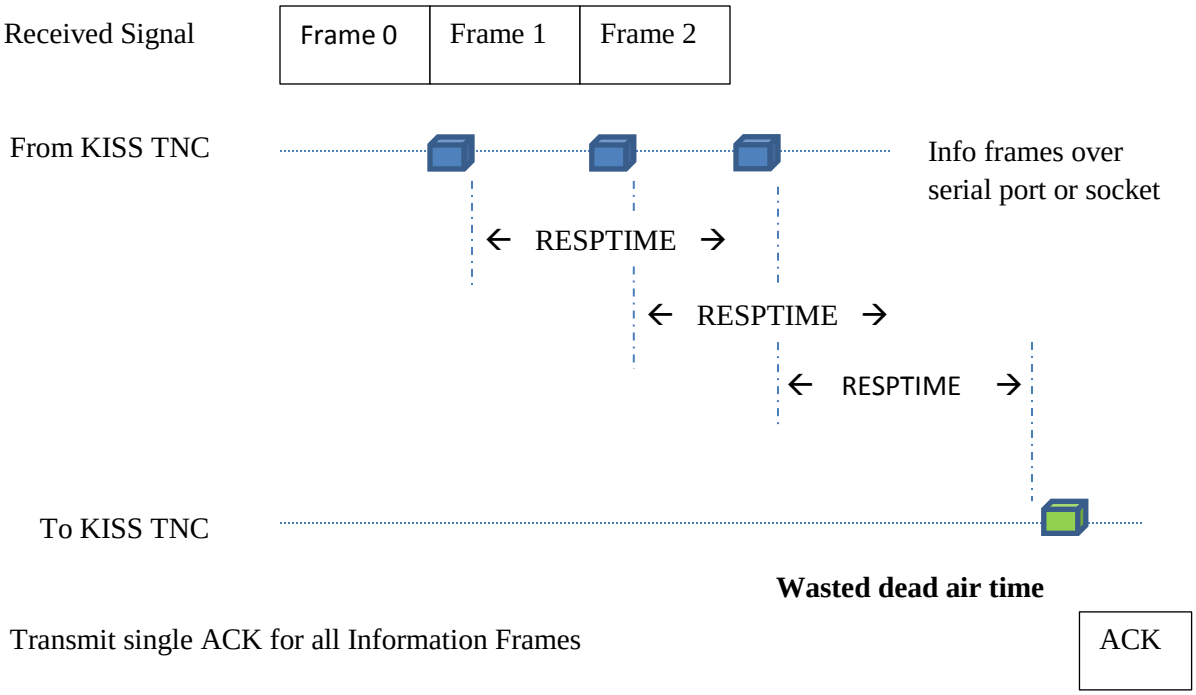
If you are using a KISS TNC, the connected mode (data link layer) software doesn't know anything about the activity on the radio channel. There is no receive modem carrier detect status available. The stupid approach would be to generate an ACK after each Information frame is received. These would pile up inside the KISS TNC until the channel is clear.



There is a kludge to cope with this situation. The earlier AX.25 protocol specification contained this:

2.4.7.1.2 Response Delay Timer T2

The second timer, T2, may be implemented by the DXE to specify a maximum amount of delay to be introduced between the time an I frame is received, and the time the resulting response frame is sent. This delay may be introduced to allow a receiving DXE to wait a short period of time to determine if there is more than one frame being sent to it. If more frames are received, the DXE can acknowledge them at once (up to seven), rather than acknowledge each individual frame. The use of timer T2 is not mandatory, but is recommended to improve channel efficiency.



In this case, the link layer processing does not have access to the channel busy status. It doesn't know the radio channel speed, or PACLEN of the other station, to calculate the maximum frame time. How does it know when it has reached the last frame in the transmission? After each received frame comes along, wait RESPTIME before assuming no more frames are in the same transmission.

Let's take a look at what this means.

Information bytes (n)	Frame bits (n+20) * 8	Seconds at 1200 bps	Seconds at 9600 bps
80	800	0.667	0.083
128	1184	0.987	0.123
256	2208	1.84	0.23

When the modem status is known, the RESPTIME parameter is unneeded. Providing this parameter is just giving people another way to "shoot themselves in the foot."

- The later (v2.2) AX.25 protocol specification removed everything related to implementation of the T2 timer.
- The earlier Kantronics KPC-3 (non Plus version) had the RESPTIME option. It was removed from the updated KPC-3+ (Plus version).

There is one case where the RESPTIME (T2) parameter would be useful. That is when you are using a KISS TNC so the Data Link State Machine has no idea if the channel is still busy or not. It's a workaround for not having the proper information to do it properly.

If you still insist on using a KISS TNC for connected mode packet, what would be a reasonable RESPTIME value? I've looked at various sample configurations and found these values:

- 0.5 second – Completely ridiculous. An 80 information byte frame at 1200 bps, is longer than this. You are going to send a separate ACK for each frame. Why bother implementing the concept?
- 1 second – Maybe for a PAC LEN of 80. Or 9600 bps.
- 2 seconds – OK for PACLEN of 128.
- 3 seconds – OK for PACLEN up to 256.
- 5 seconds – Really wasteful.

Wat would be a good value?

- You want RESPTIME to be longer than the longest possible frame from the other station so you generate a single ACK for multiple information frames in the same transmission.

- You don't want it to be larger than necessary because it will just be wasted dead air time.

10 When a frame gets lost

So far we have only been talking about the ideal case where all of the frames are received perfectly. In reality, some of them won't make it. What happens in this case?

Station A	Transmit Direction	Transmit Direction	Station B	Given to user
Information frame 0	→			I frame 0
Information frame 1	→			I frame 1
Information frame 2	→			I frame 2
Information frame 3	→ (gets clobbered)		*(not received)*	
Information frame 4	→		*(Discarded)*	
Information frame 5	→		*(Discarded)*	
		←	Got it; Ready for 3	
Information frame 3	→			I frame 3
Information frame 4	→			I frame 4
Information frame 5	→			I frame 5
		←	Got it; Ready for 6	

Suppose the first 3 information frames are received properly. These are passed along to the station operator, or some other application.

- Frame 3 gets clobbered and does not reach the destination.
- Frames 4 and 5 are received correctly but they are not expected yet. The receiving end discards them.
- At the end of the transmission, the receiving end asks for a retransmission of everything starting with the missing frame.
- Frames 3, 4, and 5 are sent again.

You are probably thinking, "This is silly." 4 & 5 were received fine. Why doesn't the receiving side hold on to them and ask for a retransmission of only 3? We will get back to this in a later section.

Let's just say that the v2.0 specification was done back when memory was small and expensive. Holding on to those out of sequence frames might not be feasible on very small resource constrained implementations.

The immediate lesson is that if the window size is very large, you run the risk of having to resend many frames again even though they got there OK the first time. (Be patient. We have a solution for this.)

11 Selective Reject

We previously observed that the downside of a very large window (MAXFRAME) value is that we could end up needlessly resending frames that already got there OK. The AX.25 v2.2 specification adds a new capability called "Selective Reject" (SREJ). The name doesn't make sense to me. Selective resend would make more sense to me. This is how it works

Station A	Transmit Direction	Transmit Direction	Station B	Given to user
Information frame 0	→			I frame 0
Information frame 1	→			I frame 1
Information frame 2	→ (gets clobbered)		*(not received)*	
Information frame 3	→		*(save 3 in memory)*	
Information frame 4	→		*(save 4 in memory)*	
Information frame 5	→ (gets clobbered)		*(not received)*	
Information frame 6	→		*(save 6 in memory)*	
		←	Resend 2 & 5	
Information frame 2	→			I frame 2
				I frame 3
				I frame 4
Information frame 5	→			I frame 5
				I frame 6
		←	Got it; Ready for 7	

This is much better.

- Frames 3 & 4 & 6 arrive out of the expected sequence so they are set aside until later.
- The receiving end figures out that 2 & 5 are missing. It asks for a resend of only those missing.
- The sending side resends only what is necessary.
- The receiving side puts it all back into the proper sequence and delivers it to the user in the right order.

With "Selective Reject" there is no longer a penalty for very large window sizes.

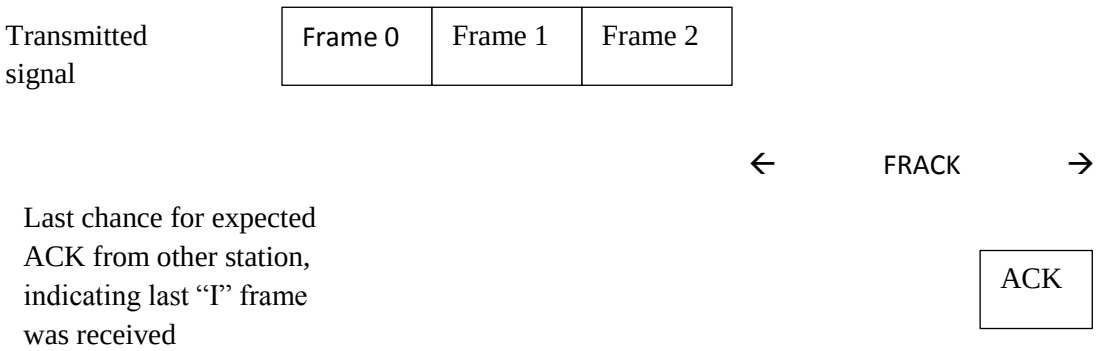
Note that only the two end nodes need to support Selective Reject for this to work. Digipeaters are only concerned with the digipeater addresses and shouldn't care whether the end nodes are using v2.0 or v2.2.

12 Timeout and retry

The other case we must consider is when the entire transmission (one or more frames) gets lost. The receiving end never got anything so it doesn't know that it needs to ask for a repeat.

Station A	Transmit Direction	Transmit Direction	Station B	Given to user
Information frame 0	→ (gets clobbered)		*(not received)*	
Information frame 1	→ (gets clobbered)		*(not received)*	
(No reply for FRACK seconds …)				
What are you expecting?	→			
		←	Ready for 0	
Information frame 0	→			I frame 0
Information frame 1	→			I frame 1
		←	Got it; Ready for 2	

After sending the last information ("I") frame in a transmission, the sender starts the FRACK (officially called the "T1") timer. If the expected ACK does not arrive before that period expires, the retry sequence is initiated. It is not simply resending the most recent data. Instead it asks the other end what is expected. The receiving end responds with what it is expecting next and the sending side rewinds and picks up from there.



The timer starts at the end of the transmission.

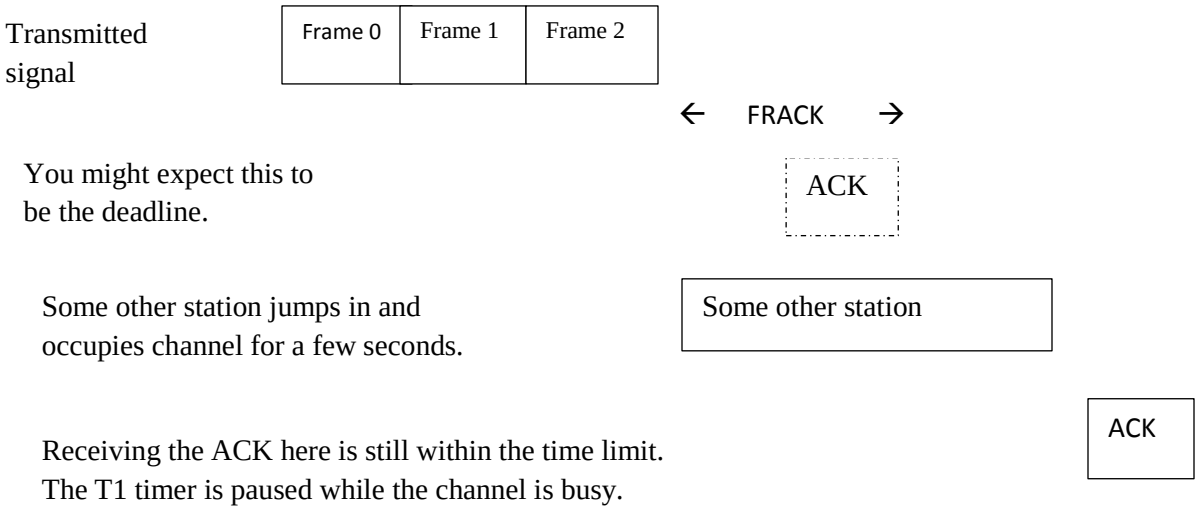
If the expected ACK, from the other station, does not arrive within FRACK seconds, the retry sequence is started.

The default value is generally 3, or sometimes 4, seconds. That gives the receiving station 2 or 3 seconds to start the reply.

What if digipeaters are involved?

The TNC is aware of this and automatically increases the internal FRACK value to be larger than the user specified value.

What if other stations jump in there?



If some other stations jump in there, it is impossible for the receiving station to send ACK within the FRACK time limit. There is a very simple solution. The timer is paused while the channel is busy. The later ACK, as seen above, still arrives before the timer expires.

Adaptive FRACK

While it would not be mandatory to implement, the v2.2 protocol spec describes how this can be adjusted automatically. The user specified value is just a starting point when a new connection is established.

The TNC keeps a running average of the ACK response times. This is used to automatically set a working value which is comfortably long enough but not overly large to be wasteful.

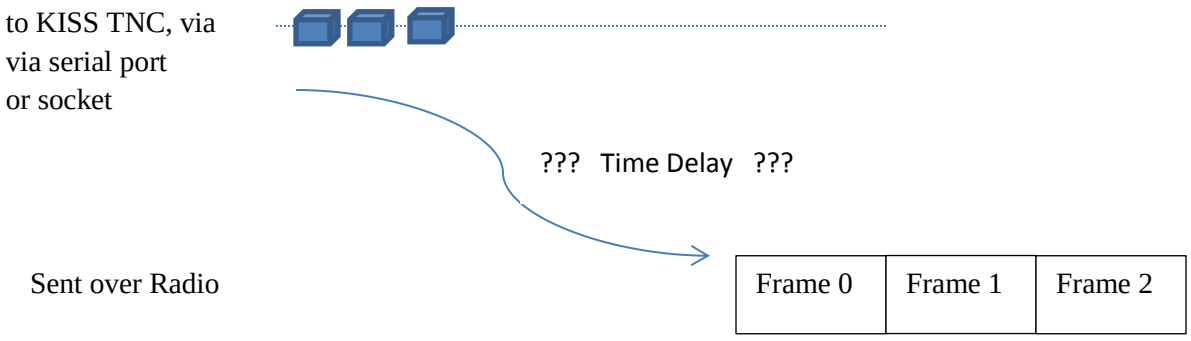
13 FRACK and the KISS TNC?

The complete TNCs usually have a default value of 3, maybe 4, seconds but some of the applications that send AX.25 link layer over a KISS TNC have sample configurations with 8 or even 10 seconds. Why so long?

As discussed in the previous section, the Link Layer processing should know about the state of the transmitter (PTT) and modem data carrier detect (DCD) to start and pause the T1 timer.

This information is not known with a KISS TNC. A request is sent to transmit some frames. A little while later you are wondering, what happened?

- Were they already sent over the radio?
- Are they still waiting in the TNC because the channel is busy?
- Were some discarded because enough memory was not available?



The higher level protocol doesn't know when to start the T1 (FRACK) timer because it doesn't know when the frames were (or will be) transmitted. It has no way of knowing whether the channel is busy with some other station so it can't pause the T1 (FRACK) timer. This is why you see up to 10 seconds in some of the sample configurations. Pick a very large value, hope for the best, and suffer the consequences.

KISS TNCs are fine for APRS where you just throw an occasional independent packet out there and don't care about the timing. Trying to run the AX.25 link layer on top of a KISS TNC can be made to work, but not well. There is not enough information available about the radio channel state.

These problems were recognized in the early 1990's and the "6PACK" protocol was developed for a closer coupling between a simple stupid (KISS) TNC and higher level protocols. This, and other similar attempts to convey more radio channel status information, never caught on and were not widely implemented.

14 AX.25 v2.0 & v2.2 compatibility

Throughout this document, there have been many references to the v2.2 revision of the AX.25 protocol standard. Unfortunately most of the existing implementations, from before that time, were never updated with the improvements from more than 20 years ago.

You might be wondering: Can the v2.0 and v2.2 versions talk to each other?

YES!

What happens when an older implementation requests a connection?

Originating Station - V2.0	Transmission direction	Transmission direction	Answering Station - Either version	Comment
SABM	→			Request v2.0 connection
		←	UA	Accepted. We will use v2.0

What happens when a newer implementation connects to newer one?

Originating Station - V2.2	Transmission direction	Transmission direction	Answering Station - V2.2	Comment
SABM**E**	→			Request v2.2 connection (note SABME vs. SABM)
		←	UA	Accepted. We will use v2.2

And, now, what you have been waiting for….

What happens when a newer implementation connects to an older one?

Originating Station - V2.2	Transmission direction	Transmission direction	Answering Station - V2.0	Comment
SABM**E**	→			Request v2.2 connection
		←	FRMR	Invalid command. (I have no idea what you are talking about.)
SABM	→			Request v2.0 connection
		←	UA	Accepted. We will use v2.0

The v2.0 protocol specification clearly states:

> ***2.3.4.3.3 Frame Reject (FRMR) Response***
>
> *2.3.4.3.3.1*
>
> *The FRMR response frame is sent to report that the receiver of a frame cannot successfully process that frame and that the error condition is not correctable by sending the offending frame again. Typically this condition will appear when a frame without an FCS error has been received with one of the following conditions:*
>
> 1. *The reception of an invalid or not implemented command or response frame.*
> 2. *[others …]*
>
> *An invalid or not implemented command or response is defined as a frame with a control field that is unknown to the receiver of this frame.*

SABME was not part of the **v2.0** protocol. When an older implementation encounters this unknown command, it replies with **FRMR**.

The **v2.2** version never sends **FRMR** under any conditions. When it receives **FRMR**, it knows it is talking to an older implementation and falls back to the older protocol version.

They are completely interoperable. If both stations understand the newer protocol version, they will benefit from the newer features. When old and new are mixed, they both use the older protocol.

15 Modem performance

One factor often overlooked, when choosing a TNC, is how well a demodulator works with less-than-ideal signals. A slightly lower decode rate could result in many lost frames and the need to send them again. A slightly lower decode rate could result in nothing getting through at all.

Suggested Reading:

- **A Better APRS Packet Demodulator, part 1, 1200 baud**

  Sometimes it's a little mystifying why an APRS / AX.25 Packet TNC will decode some signals and not others. A weak signal, buried in static, might be fine while a nice strong clean sounding signal is not decoded. Here we will take a brief look at what could cause this perplexing situation and a couple things that can be done about it.

- **A Better APRS Packet Demodulator, part 2, 9600 baud**

  In the first part of this series we discussed 1200 baud audio frequency shift keying (AFSK). The mismatch between FM transmitter pre-emphasis and the receiver de-emphasis will cause the amplitudes of the two tones to be different. This makes it more difficult to demodulate them accurately. 9600 baud operation is an entirely different animal. ...

- **WA8LMF TNC Test CD Results a.k.a. Battle of the TNCs**

  How can we compare how well the TNCs perform under real world conditions? The de facto standard of measurement is the number of packets decoded from WA8LMF's TNC Test CD. Many have published the number of packets they have been able to decode from this test. Here they are, all gathered in one place, for your reading pleasure.

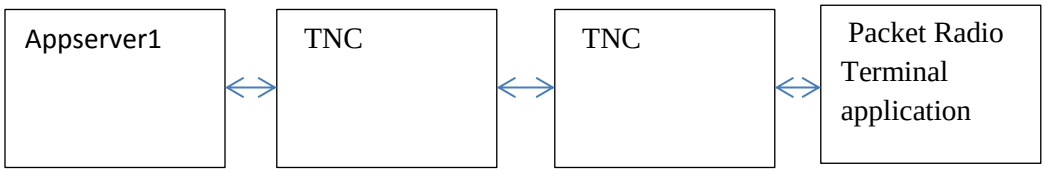
- **A Closer Look at the WA8LMF TNC Test CD**

  Here, we take a closer look at some of the frames on the TNC Test CD in hopes of gaining some insights into why some are easily decoded and others are more difficult. There are a lot of ugly signals out there. Many can be improved by decreasing the transmit volume. Others are just plain weird and you have to wonder how they are being generated.

16 Experimental Results

Now that you have heard the theory, let's try some experiments and measure the actual impact.

We will connect a packet radio terminal application, two TNCs, and a little test application together like this:



The little test application is "appserver1" from direwolf version 1.6. It is provided mostly as a starting point for writing your own server application (Bulletin Board System, Mail Transfer, etc.) using a network TNC over TCP/IP. It recognizes a command to generate a specified number of information frames with a specified size. This is to simulate a file transfer so we can measure the throughput.

We can then try tuning various parameters, such as window size, and measure the impact. We will also inject random errors and see how much that slows things down. We will explore both 1200 and 9600 bps data rates.

For the terminal application we need something that speaks the AGW network protocol. EasyTerm by UZ7HO works well. Ipagwpe, included with Outpost PM, is usable but has problems.

This was done on Windows but you should be able to do the same thing on Linux with slight configuration file changes.

16.1 Experiment 1 - 1200 bps vs. 9600 bps

For our first experiment, we will look at the relative speeds of 1200 & 9600 bps.

I find a popular e-mail application and some ARES guides recommending PACLEN 128 and MAXFRAME 2 so we will go with that to set a baseline for later comparison. We will force use of AX.25 v2.0.

	Server side	Client side
Configuration file name	direwolf-usb4.conf	direwolf-usb3.conf
Configuration file content	ADEVICE (4- MODEM 1200 AGWPORT 8004 KISSPORT 0 FIX_BITS 0 MAXFRAME 2	ADEVICE (3- MODEM 1200 AGWPORT 8000 KISSPORT 0 FIX_BITS 0 MAXFRAME 2 V20 WB2OSZ-15
Direwolf command line	direwolf -c direwolf-usb4.conf	direwolf -c direwolf-usb3.conf
Application command line	appserver1 -p 8004 WB2OSZ-15	(Terminal application with AGW network TNC interface)

On the client side, we connect to the server then type the command:

```
TEST 500 128
```

This means send back 500 information frames with 128 bytes each in the information part. This would be equivalent to transferring a file of 500 x 128 = 64000 bytes with PACLEN of 128.

Change 1200 to 9600 in the configuration files. Run the same test again.

Experiment	Speed (bps)	Frames	PACLEN	Duration (sec)	Bytes / sec	Efficiency
1	1200	500	128	1115	57	38
1	9600	500	128	499	128	10

How long would we expect it to take? Ideally, with no overhead at all, 1200 / 8 = 150 bytes per second.

At 9600, it would be 8 times that, 1200 bytes per second.

Of course we can never reach that. But if we don't get a large fraction of that, something is in serious need of improvement.

- At 1200 bps we are moving data at 38% of the channel bit rate. **Sad**
- At 9600 bps we are moving data at 10% of the channel bit rate. **Pathetic.**

The data bit rate was increased by a factor of 8 but the throughput is only about twice as fast.

16.2 Experiment 2 – Larger PACLEN

Keep everything the same. This time we will use this command instead:

```
TEST 250 256
```

This is the same amount of data. It is just split up differently as half as many frames and doubling the information part size.

Results compared with Experiment 1

Experiment	Speed (bps)	Frames	PACLEN	Duration (sec)	Bytes / sec	Efficiency
1	1200	500	128	1115	57	38
2		250	256	767	83	55
1	9600	500	128	499	128	10
2		250	256	258	248	20

That's a very significant improvement. PACLEN 128 on VHF FM doesn't make much sense.

16.3 Experiment 3 – Larger MAXFRAME

In this experiment we will test the impact of changing the window size. This means more information frames can be sent before waiting for the acknowledgement to continue.

Rather than the original 2, we will try 4 and 7. Change the lines highlighted in red below.

	Server side	Client side
Configuration file name	direwolf-usb4.conf	direwolf-usb3.conf
Configuration file content	ADEVICE (4- MODEM 1200 AGWPORT 8004 KISSPORT 0 FIX_BITS 0 **MAXFRAME 7**	ADEVICE (3- MODEM 1200 AGWPORT 8000 KISSPORT 0 FIX_BITS 0 **MAXFRAME 7** V20 WB2OSZ-15
Direwolf command line	direwolf -c direwolf-usb4.conf	direwolf -c direwolf-usb3.conf
Application command line	appserver1 -p 8004 WB2OSZ-15	(Terminal application with AGW network TNC interface)

Experiment	Speed (bps)	MAXFRAME	Duration (sec)	Bytes / sec	Efficiency
2	1200	2	767	83	55
3		4	611	104	69
3		7	543	117	78
2	9600	2	258	248	20
3		4	163	392	32
3		7	122	524	43

Larger MAXFRAME is more efficient. We are doing well at 1200 bps but still not so good at 9600 bps.

There is a tradeoff for unreliable radio links. We will get to that in the next section.

16.4 Experiment 4 – Unreliable radio links, v2.0

So far we have considered only the cases where all of the frames get received properly. What happens when something gets lost along the way and data needs to be resent?

In this experiment we will intentionally introduce errors. The direwolf "-E" command line option will cause the specified percentage of transmitted frames to be corrupted so the CRC does not match and the receiving station will discard it. We will look at the impact of both 5 and 10 percent frame loss in each direction.

We will continue to use MAXFRAME 7, as in the previous experiment.

Again use "TEST 250 256" for all cases.

	Server side	Client side
Configuration file name	direwolf-usb4.conf	direwolf-usb3.conf
Configuration file content	ADEVICE (4- MODEM 1200 AGWPORT 8004 KISSPORT 0 FIX_BITS 0 MAXFRAME 7	ADEVICE (3- MODEM 1200 AGWPORT 8000 KISSPORT 0 FIX_BITS 0 MAXFRAME 7 V20 WB2OSZ-15
Direwolf command line	direwolf -c direwolf-usb4.conf **–E 10**	direwolf -c direwolf-usb3.conf **–E 10**
Application command line	appserver1 -p 8004 WB2OSZ-15	(Terminal application with AGW network TNC interface)

Results: Here we copied experiment 3 results for easy comparison. As you would expect, when some of the frames get clobbered and have to be sent again, the overall throughput decreases.

Experiment	Speed (bps)	Error Rate % corrupt	Duration (sec)	Bytes / sec	Efficiency
3	1200	0	543	117	78
4		5	620	103	68
4		10	715	89	59
3	9600	0	122	524	43
4		5	172	372	31
4		10	230	278	23

This should demonstrate why you should care about the quality of the modem. The TNC Test CD has over 1000 AX.25 frames of varying qualities. A TNC that can decode 950 has a loss rate of about 5%. A TNC that can decode 900 has a loss rate of about 10%.

16.5 Experiment 5 – Selective Reject

The error recovery in the v2.0 protocol spec was very wasteful. Recall that any information frames received out of the expected sequence were simply discarded. The receiving end asks for retransmission of everything starting with the first missing frame. A larger window size means potentially retransmitting more frames redundantly.

V2.2 added a new feature that saves the out-of-sequence frames, asks for only those missing, and puts it all into the proper order.

Here we will enable the version 2.2 protocol. We simply remove the V20 line from the configuration file and repeat the previous experiment.

	Server side	Client side
Configuration file name	direwolf-usb4.conf	direwolf-usb3.conf
Configuration file content	ADEVICE (4- MODEM 1200 AGWPORT 8004 KISSPORT 0 FIX_BITS 0 MAXFRAME 7 EMAXFRAME 7	ADEVICE (3- MODEM 1200 AGWPORT 8000 KISSPORT 0 FIX_BITS 0 MAXFRAME 7 EMAXFRAME 7 ~~V20 WB2OSZ-15~~
Direwolf command line	direwolf -c direwolf-usb4.conf **–E 10**	direwolf -c direwolf-usb3.conf **–E 10**
Application command line	appserver1 -p 8004 WB2OSZ-15	(Terminal application with AGW network TNC interface)

Result: Resending the missing frames is now more efficient.

			V2.0 Implicit Reject			V2.2 Selective Reject		
Experiment	Speed (bps)	Error Rate % corrupt	Duration (sec)	Bytes / sec	Efficiency	Duration (sec)	Bytes / sec	Efficiency
3	1200	0	543	117	78			
4, 5		5	620	103	68	613	104	69
4, 5		10	715	89	59	697	91	61
3	9600	0	122	524	43			
4, 5		5	172	372	31	155	412	34
4, 5		10	230	278	23	191	335	27

Now that we have enabled v2.2 features, let's see what happens with MAXFRAME 63.

16.6 Experiment 6 – Increasing efficiency for 9600

We have seen how increasing the frame size and window size allowed the efficiency to improve. This allowed a larger portion of the time to be used for the useful data and less on overhead.

9600 is still lagging far behind because the data is sent so quickly the overhead is still rather high. 7 frames of 256 takes about a second and a half.

We already reached the limits for AX.25 v2.0. We can only get around 4 times the throughput with 8 times the bit rate.

Another 9600 vs. 1200 study came up with a similar ratio (slide 42). In this case, about 425 vs. 100 bytes/sec.

AX.25 v2.2 increases these limits. Let's try doubling the frame size again. Then we will try a window size of 32 (default) and 63 (maximum).

	Server side	Client side
Configuration file name	direwolf-usb4.conf	direwolf-usb3.conf
Configuration file content	ADEVICE (4- MODEM 1200 AGWPORT 8004 KISSPORT 0 FIX_BITS 0 PACLEN 512 EMAXFRAME 63	ADEVICE (3- MODEM 1200 AGWPORT 8000 KISSPORT 0 FIX_BITS 0 PACLEN 512 EMAXFRAME 63 ~~V20 WB2OSZ-15~~
Direwolf command line	direwolf -c direwolf-usb4.conf	direwolf -c direwolf-usb3.conf
Application command line	appserver1 -p 8004 WB2OSZ-15	(Terminal application with AGW network TNC interface)

Here we use half as many and twice the size. This cuts number of ACKs in half. EMAXFRAME is varied between tests.

```
TEST 125 512
```

Experiment	Speed (bps)	PACLEN	Window EMAXFRAME	Duration (sec)	Bytes / sec	Efficiency
3	1200	256	7	543	117	78
6		512	7	515	124	82
		512	32	474	135	90
		512	63	467	137	91
3	9600	256	7	122	524	43
6		512	7	85	752	62
		512	32	63	1015	84
		512	63	60	1066	88

We finally get respectable throughput efficiency for 9600 bps.

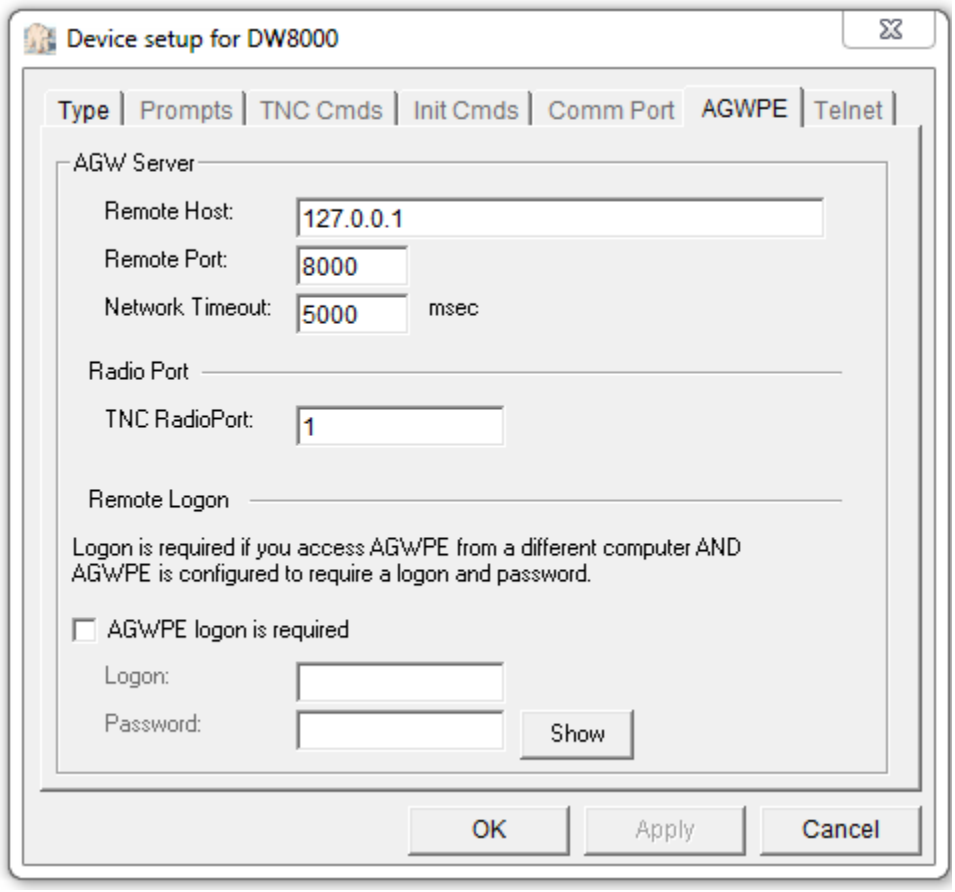
For 9600 bps, the AX.25 v2.2 protocol can double the throughput possible with v2.0.

16.7 Experiment 7 - KISS TNC rather than AGW network interface

Throughout this document, we pointed out numerous ways that trying to run the AX.25 link layer with a KISS TNC is not such a good idea. Yes, it can be made to work, but with work-arounds, compromises, and reduced performance.

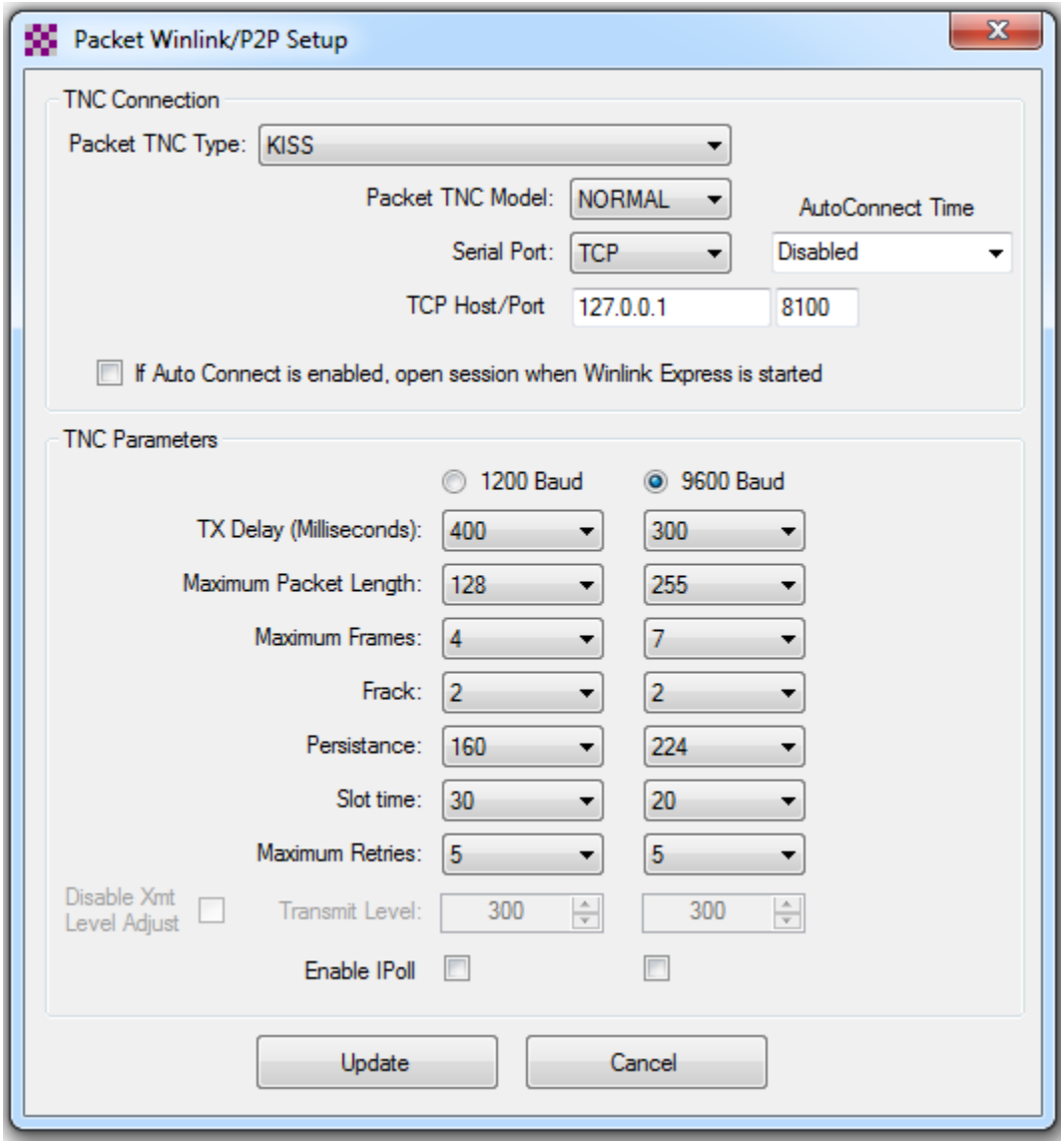
My goal, for this section, is to take some popular messaging applications, such as **Outpost PM** or **WinLink Express**, and compare performance with the KISS protocol vs. AGW network interface.

Outpost PM has the ability to use an AGW network interface. This is good.



However it doesn't seem to have KISS over TCP. I guess I will need to set up a couple "null modem cable" virtual COM port pairs to connect them to each other. Let's get back to this later.

How about WinLink Express? If we select “Packet P2P” it has the ability to use KISS over TCP.



From page 20 here, https://winlink.org/content/winlink_faq_sept_15_2018_frequently_asked_questions_answers the older Paclink supported the network TNC protocol but that capability hasn’t made it into WinLink Express yet.

To be continued….

17 Summary

It is easy to find lots of information about Amateur Packet Radio operation. Unfortunately most of it is terribly outdated and often wrong. Example: A fairly recent ARES presentation, which I stumbled across, claims that you need to buy a $400 TNC to use 9600 baud packet radio. ☹ That's enough to scare most people away. The truth is that you can get better results with free software.

The background and techniques, presented here, should help you spot poor practices and improve the efficiency of your Packet Radio data transfers while we are still struggling to cram data through radios designed for voice.

Key takeaways:

- 9600 bps Packet Radio will not work with the microphone and speaker connections because the transmitter and receiver intentionally distort the audio. Good for voice, bad when trying to user with a modem. You will want to use a transceiver with a special connector that allows you to bypass these audio processing circuits.

- A larger PACLEN is more efficient. This allows fewer frames and fewer acknowledgements for the same amount of useful data.

- A larger MAXFRAME (or EMAXFRAME) value is more efficient. More frames can be sent in a single transmission before stopping and waiting for a reply.

- Avoid running the AX.25 link layer on top of a KISS TNC. It doesn't provide enough information about the radio channel state for the higher level protocol to make good decisions. Use the implementation built into a traditional TNC or the AGW protocol for a network attached TNC.

- The AX.25 v2.2 standard (now over 20 years old) has several improvements to make bulk data transfer more efficient. It is fully interoperable with the old 1984 standard so everyone can get along. This is the key to unlocking the potential of 9600 bps.

Speed, bits per second (bps)	Theoretical maximum bytes per second at 100% efficiency (unobtainable)	Useful Throughput (bytes per second) **AX.25 v2.0**	Useful Throughput (bytes per second) **AX.25 v2.2**
1200	150	117	137
9600	1200	524	1066

Someday we will finally break free of 20th Century technology and use more advanced modulation techniques and radios more friendly to sending digital data. Until then, we can make significant efficiency improvements while maintaining compatibility with the large installed base.