Improved Layer 2 Protocol

Nino Carrillo KK4HEJ

IL2P Overview

IL2P is a Layer 2 packet format that incorporates Forward Error Correction (FEC), packet-synchronous scrambling, and efficient encoding of variable-length packets for narrow-band digital data links. IL2P builds on the extensive work done by others in the amateur radio field to improve the quality, speed, and flexibility of packet radio data networks. IL2P is inspired and informed by the FX.25 draft standard, but departs from on-air backwards compatibility with AX.25 in order to implement a more capable standard. Several of the IL2P Design Goals stem directly from recommendations made by the authors of FX.25 in their draft specification document.

Initial implementations of IL2P target compatibility with the standard AX.25 KISS interface to transfer data to and from a local host device. Many popular host applications (like linBPQ and APRS servers) expect TNCs to speak AX.25 KISS. Therefore, the first hardware implementation of IL2P in existence translates AX.25 KISS frames into IL2P for broadcast on-air, and converts them back to AX.25 KISS frames at the receive side to send them to the host.

Cost of custom-made printed circuit boards and fast embedded digital signal processors are significantly lower today than in 2006, when the FX.25 draft standard was published. It now is possible to implement a KISS TNC in low-power embedded firmware that can encode and decode IL2P packets in real time, while listening for legacy AX.25 packets, and performing 1200 baud AFSK or 9600 baud GFSK modulation and demodulation on a datastream. It is the author’s hope that these hybrid firmware TNCs, which can offer legacy AX.25 compatibility in parallel with IL2P capabilities at lower cost than traditional hardware TNCs, accelerate the adoption of this improved standard.

Design Goals

- Incorporate forward-error-correction
- Eliminate bit-stuffing
- Streamline the AX.25 header format
- Improve packet detection in absence of DCD and for open-squelch receive
- Produce a bitstream suitable for modulation on various physical layers
- Avoid bit-error-amplifying methods (differential encoding and free-running LFSRs)
- Increase efficiency and simplicity over FX.25

Interface to Physical Layer

IL2P can be applied to various modulation methods including Audio Frequency Shift Keying (AFSK), Gaussian Frequency Shift Keying (GFSK), and any others that support binary symbols. A '1' bit in IL2P is sent as an AFSK "mark" tone (1200 Hz), while a '0' bit is sent as an AFSK "space" tone (2200Hz). When using 9600 GFSK, a '1' bit is sent as positive FM carrier deviation (appears as a positive voltage pulse on the TNC’s TXA line), and a '0' bit is sent as negative FM carrier deviation. Unlike Bell 202 Non-Return-to-Zero Inverted (NRZI) AFSK and G3RUH 9600, IL2P **does not** use differential encoding.

Technical Details

Reed Solomon Forward Error Correction

Reed-Solomon (RS) forward-error-correction is used to detect and correct errors in the header and payload blocks. The IL2P RS encoder processes header and payload data *after* it has been scrambled, to eliminate the error-amplifying characteristics of multiplicative LFSRs. RS codes have maximum block lengths defined by their underlying Galois Field (GF) size. IL2P uses an 8-bit field to match the size of a byte. The Galois Field is defined by reducing polynomial x^8+x^4+x^3+x^2+1. The maximum RS block size is 255 bytes, including parity. In order to support packets larger than the RS block size, large packets are segmented by the encoder into nearly-equal sized blocks before RS encoding into a contiguous IL2P packet.

Variable parity lengths of 2, 4, 6, 8, or 16 symbols (bytes) are used depending on the size of the payload block and selected FEC strength. This allows for increased efficiency for short packets, and provides a consistent symbol-error capability independent of packet length. Variable code shortening is used to eliminate block padding, enabled by a payload byte count subfield in the header.

The RS encoder uses zero as its first root.

IL2P does not use a Cyclic Redundancy Check (CRC) or Frame Check Sequence (FCS). Instead, validity of received data is verified through successful decoding of the RS blocks.

Data Scrambling

IL2P employs packet-synchronous multiplicative scrambling to reduce transmit signal occupied bandwidth, ensure sufficient zero crossings for the receive data-clock PLL, and DC-balance the transmit bitstream. The scrambling is carried out by a linear-feedback-shift-register (LFSR), using feedback polynomial x^9+x^4+1, which is maximal. This polynomial is significantly lower-order than that used in G3RUH 9600 modems. Selection of a lower order ensures the longest runs of continuous 1 or 0 bits will be shorter, which aids receive data-clock stability.

Packet-Synchronized LFSR

The LFSR is reset to initial conditions at the start of every packet. Scrambling begins at the first bit after the Sync Word. The Preamble and Sync Word are not scrambled. During receive, prior to Sync Word detection, the LFSR is not engaged. The LFSR state is unaltered between blocks inside a packet, scrambling or unscrambling continues with the state left at the end of the last block.

Scrambling Inside RS Code Block

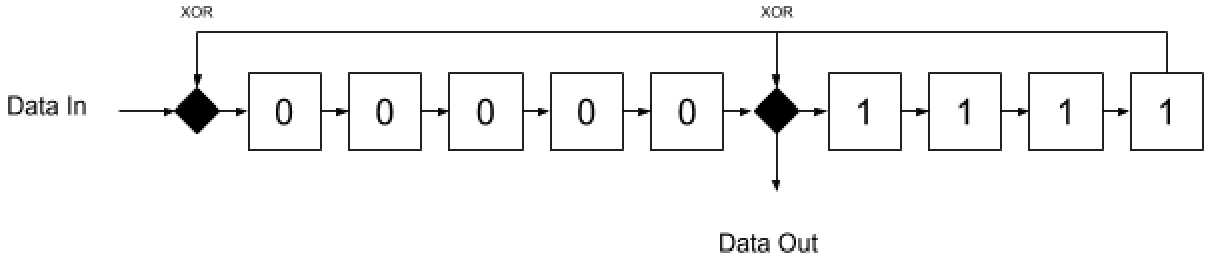
IL2P LFSR encoding is applied inside the RS code block to eliminate the bit-error-amplifying characteristics of LFSR processing. A free-running LFSR (such as in the receive circuitry of the G3RUH modem) propagates bit errors at a multiple of the number of feedback polynomial coefficients (or taps on the LFSR). For example, when a single bit-error passes through a free-running LFSR defined by X^9+X^4+1 (or any other 3-term polynomial), 3 erroneous bits will appear on the output as they are XOR'd through the feedback taps of the shift register. This is of little concern in legacy AX.25 on-air protocols, because even a single bit error anywhere in the packet will cause the packet to be rejected.

RS codes correct errors on a symbol-by-symbol basis (byte-by-byte for IL2P). In order to prevent the LFSR spreading a single bit error from one RS symbol to another, the IL2P packet encoder applies RS encoding *after* the data has been scrambled, and the receiver applies RS decoding *before* the data is unscrambled. This allows bit errors to be corrected by the RS decoder before passing through the receive LFSR. The RS parity symbols themselves are not passed through an LFSR, they are appended to the RS block exactly as computed.

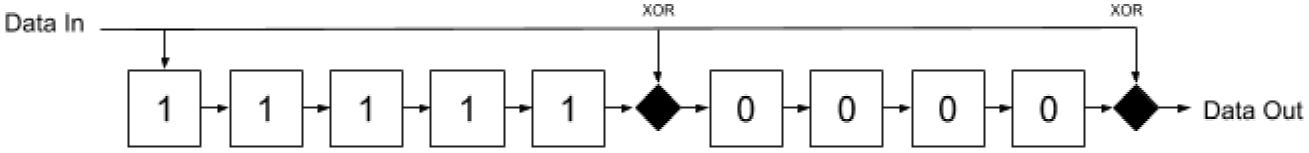
Extracting All Data from LFSR Memory

Efficient LFSR algorithms can be constructed by arranging an LFSR in Galois configuration. Galois configured LFSRs have bit delay, which means it takes some number of bit cycles after a bit of information enters the LFSR for it to appear in its scrambled form on the output. Because of this, the output of the LFSR is taken after its bit delay has elapsed (5 bits in this case), and flushed at the end of the data block to extract all information bits from its memory. The LFSR schematics given below represent Galois configuration of the IL2P scrambling polynomial.

Transmit LFSR Schematic and Initial Conditions



Receive LFSR Schematic and Initial Conditions



Packet Structure

IL2P Packet Format				
Preamble	Sync Word	Control & Addressing	Header Parity	Payload Blocks & Parity
variable	3 bytes	13 bytes	2 bytes	0-1081 bytes

All bytes are sent **Most Significant Bit first**.

Preamble

The IL2P recommended Preamble is variable length, and consists of some number of 0x55 bytes (01010101), which provides the receive data slicer frequent bit transitions to establish a lock on the transmitted data-clock before information appears. When sent back-to-back, the Preamble of subsequent packets is omitted. There is no terminating symbol. All IL2P packets are terminated by byte count, which is stored in the header.

Sync Word

The IL2P Sync Word is 0xF15E48. This 24 bit sequence has an equal number of 1's and 0's and identifies the start of all IL2P Packets. Recommended Sync Word match tolerance at the receiver is 1 bit, meaning the receiver will declare a match if 23 out of the last 24 bits received match the Sync Word (any single bit flipped). This intended to ensure Sync Word detection on noisy links, at the cost of increasing the Sync Word match space up to 25 possible matches out of 2^24 possible bit sequences. In a 9600 bit/sec application with open squelch and ignoring DCD, the expected average interval time between false matches is about 69 seconds (bit rate * 2^24 / 25). False matches are rejected by the receiver after the header fails RS decoding.

FEC Level

A one-bit subfield in the header identifies the amount of FEC parity bytes applied to the packet. A zero value indicates variable FEC up to 8 bytes per block (referred to Baseline FEC in this document). A one value indicates constant FEC of 16 bytes per block (referred to as Max FEC).

IL2P Header Types

IL2P defines 2 possible header mappings, encoded in a 1-bit header subfield. A zero value indicates transparent encapsulation. A one value indicates translated encapsulation. Both mappings include a 10-bit payload count, enabling packet sizes up to 1023 payload bytes after the header. This count does not include parity bytes attached to the payload.

IL2P Type 0 Header

Type 0 headers are used for transparent encapsulation of data - the entire encapsulated packet appears in the payload of the IL2P packet. Therefore, the header only includes the 10 bit PAYLOAD BYTE COUNT subfield as described in IL2P Type 1 Header. Type 0 encapsulation occurs when a KISS frame is presented to the IL2P encoder that cannot be translated. Some examples of non-translatable KISS frames include MIC-E encoded APRS data (callsign characters can't translate to SIXBIT), Extended mode AX.25 frames (modulo-127 window sizes), and unrecognized AX.25 PID codes. These frames are placed entirely in the IL2P payload, so they still benefit from forward-error-correction.

IL2P Type 1 Header

Type 1 headers contain a compressed and translated AX.25 header. The majority of common AX.25 traffic is compatible with Type 1 translation. The Control and Addressing section of the header contains everything normally found in an AX.25 header, with some modifications. IL2P stores destination and source callsigns using six bits per character in DEC SIXBIT coding (take the ASCII code for a printable character and subtract 0x20). IL2P also compresses the Protocol ID field to 4 bits rather than 8.

<table>
<tr><th colspan="14">Control and Addressing Field Map for IL2P Type 1 Header</th></tr>
<tr><th></th><th>Byte 0</th><th>Byte 1</th><th>Byte 2</th><th>Byte 3</th><th>Byte 4</th><th>Byte 5</th><th>Byte 6</th><th>Byte 7</th><th>Byte 8</th><th>Byte 9</th><th>Byte 10</th><th>Byte 11</th><th>Byte 12</th></tr>
<tr><td>Bit 0</td><td rowspan="6">DEST C/S 1</td><td rowspan="6">DEST C/S 2</td><td rowspan="6">DEST C/S 3</td><td rowspan="6">DEST C/S 4</td><td rowspan="6">DEST C/S 5</td><td rowspan="6">DEST C/S 6</td><td rowspan="6">SRC C/S 1</td><td rowspan="6">SRC C/S 2</td><td rowspan="6">SRC C/S 3</td><td rowspan="6">SRC C/S 4</td><td rowspan="6">SRC C/S 5</td><td rowspan="6">SRC C/S 6</td><td rowspan="4">SRC SSID</td></tr>
<tr><td>Bit 1</td></tr>
<tr><td>Bit 2</td></tr>
<tr><td>Bit 3</td></tr>
<tr><td>Bit 4</td><td rowspan="4">DEST SSID</td></tr>
<tr><td>Bit 5</td></tr>
<tr><td>Bit 6</td><td>UI</td><td colspan="4">PID</td><td colspan="7">CONTROL</td></tr>
<tr><td>Bit 7</td><td>FEC LEVEL</td><td>HDR TYPE</td><td colspan="10">PAYLOAD BYTE COUNT</td></tr>
<tr><td colspan="14">Subfields spanning Bit 6 and Bit 7 have MSB on the left. SSID are four-bit subfields.
Callsigns are packed in DEC SIXBIT encoding.</td></tr>
</table>

Type 1 Header Control and Addressing Subfields

The Type 1 Header is composed of several fields found in the AX.25 header, though they are translated and compressed into an IL2P format. Type 1 Headers do not support AX.25 repeater callsign addressing, Modulo-127 extended mode window sequence numbers, nor any callsign characters that cannot translate to DEC SIXBIT. If these cases are encountered during IL2P packet encoding, the encoder switches to Type 0 Transparent Encapsulation.

Payload Byte Count Subfield

The Payload Byte Count is stored in the header as a 10-bit subfield (possible values 0-1023). The count represents the total number of data bytes stored in all payload blocks following the header. The count excludes the header, and all parity symbols appended to payload blocks. See the Payload Blocks section of this document for a description of how payload parity symbols are appended to payload blocks.

UI Subfield

AX.25 specifies 3 types of frames: Information, Supervisory, and Unnumbered. Each has different uses for the AX.25 Control field, and only some have a PID field. All AX.25 Information frames have a PID field. AX.25 Supervisory frames do not have a PID field. AX.25 Unnumbered frames do not have a PID field, unless their Control field is set to the Unnumbered Information (UI) opcode. The IL2P Type 1 Header UI subfield is 1 bit and is set only for AX.25 Unnumbered Information frames to signal that the PID field exists for a U-Frame.

PID Subfield

In Type 1 header mapping, IL2P maps the AX.25 8-bit PID field into a 4-bit IL2P subfield. The IL2P PID subfield is also used to identify the AX.25 frame type, which informs the encoding and decoding of the IL2P Control subfield.

IL2P AX.25 PID Code Mapping

IL2P PID	Translation	AX.25 PID
0x0	AX.25 Supervisory Frame (No PID byte)	Omit PID
0x1	AX.25 Unnumbered Frame (No PID byte, except UI)	Omit PID
0x2	AX.25 Layer 3	yy10yyyy or yy01yyyy
0x3	ISO 8208 / CCIT X.25 PLP	0x01
0x4	Compressed TCP/IP	0x06
0x5	Uncompressed TCP/IP	0x07
0x6	Segmentation fragment	0x08
0x7	Future	
0x8	Future	
0x9	Future	
0xA	Future	
0xB	ARPA Internet Protocol	0xCC
0xC	ARPA Address Resolution	0xCD
0xD	FlexNet	0xCE
0xE	TheNET	0xCF
0xF	No L3	0xF0

Control Subfield

The Control Subfield contains 7 bits, and its mapping depends on the translated AX.25 frame type.

Translated AX.25 I-Frame Control Subfield

All AX.25 I-Frames are considered commands. Therefore, IL2P omits the Command (C) bit for translated I-Frames. This subfield contains a Poll/Final (P/F) bit, receive sequence N(R), transmit sequence N(S).

Translated AX.25 I-Frame Control Subfield Map						
Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
P/F	N(R)			N(S)		

Translated AX.25 S -Frame Control Subfield

AX.25 S-Frames can be one of 4 opcodes. All include a receive sequence number N(R), and a C bit.

Translated AX.25 S-Frame Control Subfield Map						
	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	N(R)			C	OPCODE	
RR Receive Ready	N(R)			C	0	0
RNR Receive Not Ready	N(R)			C	0	1
REJ Reject	N(R)			C	1	0
SREJ Selective Reject	N(R)			C	1	1

Translated AX.25 U-Frame Control Subfield

AX.25 U-Frames contain an opcode, P/F bit, and C bit. Certain opcodes are always commands or responses, some can be either. There are no sequence numbers in U-Frames.

Translated AX.25 U-Frame Control Subfield Map								
		Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
		P/F	OPCODE			C		
command	SABME set async balanced mode extended	Not supported, send as Transparent						
command	SABM set async balanced mode	P	0	0	0	1		
command	DISC disconnect	P	0	0	1	1		
response	DM disconnect mode	F	0	1	0	0		
response	UA unnumbered acknowledge	F	0	1	1	0		
response	FRMR frame reject	F	1	0	0	0		
either	UI unnumbered information	P/F	1	0	1	C/R		
either	XID exchange identification	P/F	1	1	0	C/R		
either	TEST	P/F	1	1	1	C/R		

Payload Blocks

Each payload block forms a contiguous RS code block once parity is added. RS codes can correct a number of erroneous symbols in a code block equal to half the number of parity symbols. So a code block with 2 parity symbols can recover one erroneous symbol anywhere in the code block. Baseline FEC block lengths and parity counts in IL2P are designed to provide roughly 1.5% symbol-error-rate recovery in the payload blocks. The number of parity symbols added to each block varies based on the size of the block. To achieve that, the following procedure is conducted by the transmitter to calculate the number of payload blocks and parity symbols required to compose the packet:

Baseline FEC Payload Block Size Computations

payload_block_count = Ceiling(payload_byte_count / 247)
small_block_size = Floor(payload_byte_count / payload_block_count)
large_block_size = small_block_size + 1
large_block_count = payload_byte_count - (payload_block_count * small_block_size)
small_block_count = payload_block_count - large_block_count

Large blocks are 1 byte bigger than small blocks. Not every packet requires large blocks, they exist to carry remainder bytes. If small_block_size divides evenly into payload_byte_count, then the packet can be encoded without large blocks. Large blocks, if they exist, are always placed closest to the header when the packet is assembled.

Worked examples:

IL2P Baseline FEC Payload Block Count Examples				
Payload Byte Count	100	236	512	1023
Small Block Size	100	236	170	204
Large Block Size	101	237	171	205
Large Block Count	0	0	2	3
Small Block Count	1	1	1	2

Baseline FEC Parity Symbol Count Computation

The number of parity symbols appended to each payload block is driven by small_block_size.

parity_symbols_per_block = (small_block_size / 32) + 2

The encoder will append 2, 4, 6, or 8 parity symbols per payload block. The maximum small_block_size for each parity symbol count is given below.

Maximum small_block_size	
Parity Symbols per Block	Maximum small_block_size
2	61
4	123
6	185
8	247

Max FEC Payload Block Size Computations

Under the Max FEC scheme, the encoder will always append 16 parity symbols per payload block, regardless of block size. This provides a minimum of roughly 3% symbol-error-rate recovery in the payload blocks. Shorter packets benefit from higher error recovery capacity.

payload_block_count = Ceiling(payload_byte_count / 239)
small_block_size = Floor(payload_byte_count / payload_block_count)
large_block_size = small_block_size + 1
large_block_count = payload_byte_count - (payload_block_count * small_block_size)
small_block_count = payload_block_count - large_block_count
parity_symbols_per_block = 16

IL2P Transmit Encoding Procedure for AX.25 KISS Data

1. Place Sync Word in the first three bytes of output buffer
2. Extract all AX.25 header fields
3. Check AX.25 header fields for compatibility with Type 01 Header

If AX.25 Fields Type 1 Compatible

4. Compose IL2P Control & Addressing Field and place in output buffer
5. Initialize LFSR to initial conditions
6. Scramble the output buffer starting at the Control & Addressing Field
7. RS Encode output buffer starting at the Control & Addressing Field
8. Count payload bytes in AX.25 input data and perform Payload Block Size computations
9. Perform Parity Symbol Count computation
10. Scramble then RS encode each payload block (large blocks closest to header)
11. Send output buffer data to transmitter (AFSK or GFSK modulator)

If AX.25 Fields Not Type 1 Compatible Send As Type 0

4. Count all bytes in AX.25 input data and perform Payload Block Size computations
5. Perform Parity Symbol Count computation
6. Place PAYLOAD BYTE COUNT subfield in Control & Addressing Field (all other fields 0)
7. Scramble the output buffer starting at the Control & Addressing Field
8. RS Encode output buffer starting at the Control & Addressing Field
9. Scramble then RS encode each payload block (large blocks closest to header)
10. Send output buffer data to transmitter (AFSK or GFSK modulator)

IL2P Receive Decoding Procedure for KISS AX.25 Data

1. Search receive bitstream for Sync Word match

On Sync Word Match Within 1 Bit Tolerance

2. Collect next 15 bytes as IL2P Header
3. RS Decode IL2P Header

If RS Decode Successful

4. Initialize LFSR to initial conditions
5. Unscramble 13 byte Control & Addressing Field
6. Extract IL2P Control & Addressing Field and translate to AX.25 header in KISS buffer
7. Perform Payload Block Size computations on PAYLOAD BYTE COUNT
8. Perform Parity Symbol Count computation
9. Collect payload blocks from receive bitstream according to results of Step 7 and 8
10. RS decode and then unscramble each payload block
11. Place unscrambled data in KISS buffer and send to host
12. Return to Step 1

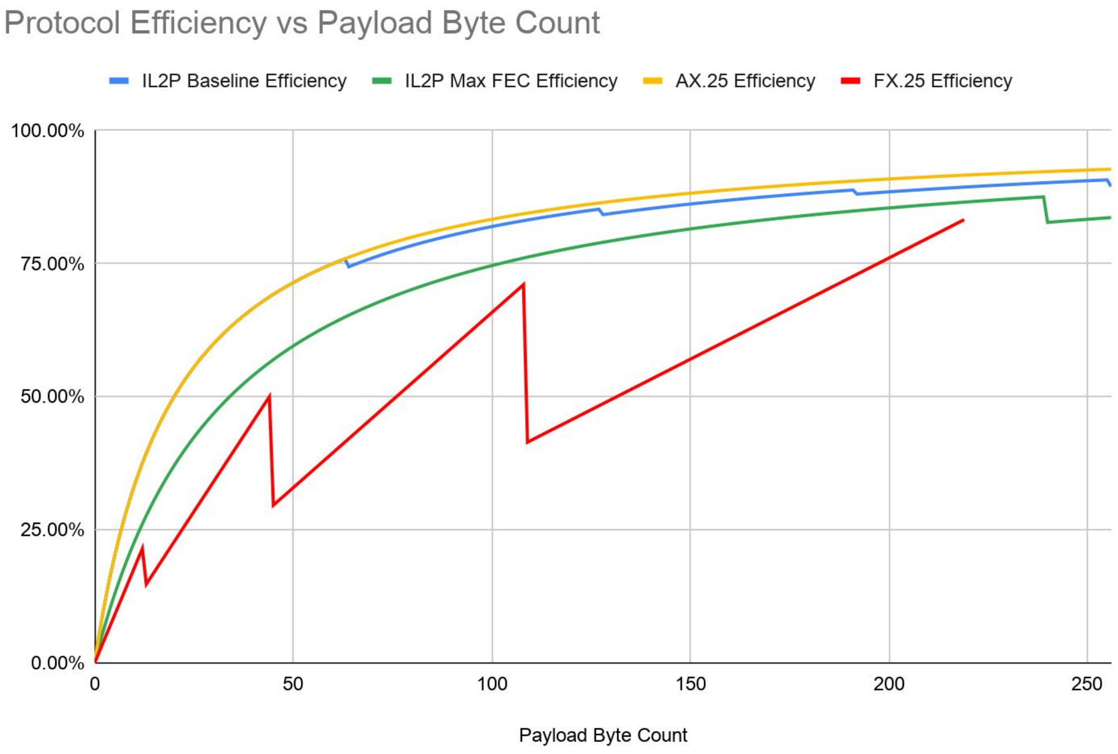
If RS Decode of Header or Any Payload Block Unsuccessful

13. Discard packet
14. Return to Step 1

Comparative Protocol Efficiency Analysis

Protocol Efficiency in the graph below shows the percentage of payload bytes that make up the packet, excluding Preamble. The IL2P Header and Sync Word consume 18 bytes, so efficiency generally increases as packet size grows. The sawtooth bumps in the graph represent Payload Byte Counts where an additional code block is required to contain the payload.

For comparison, the efficiency of AX.25 and FX.25 (255,239) protocols are included on the graph. The FX.25 line is computed using the smallest block size compatible with the payload size. The costs of bit-stuffing incurred under AX.25 and FX.25 are ignored.



Example Encoded Packets

These examples are intended for use as verification samples to help individuals implementing their own IL2P encoders and decoders. Note that all AX.25 data samples below lack opening and closing flags, and are not bit-stuffed. All IL2P data samples below lack Sync Word.

AX.25 S-Frame

This frame sample only includes a 15 byte header, without PID field.
Destination Callsign: `KA2DEW-2`
Source Callsign: `KK4HEJ-7`
N(R): 5
P/F: 1
C: 1
Control Opcode: 00 (Receive Ready)

AX.25 data:

```
96 82 64 88 8a ae e4 96 96 68 90 8a 94 6f b1
```

IL2P Data Prior to Scrambling and RS Encoding:

```
2b a1 12 24 25 77 6b 2b 54 68 25 2a 27
```

IL2P Data After Scrambling and RS Encoding:

```
26 57 4d 57 f1 96 cc 85 42 e7 24 f7 2e 8a 97
```

AX.25 U-Frame

This is an AX.25 Unnumbered Information frame, such as APRS.
Destination Callsign: `CQ    -0`
Source Callsign: `KK4HEJ-15`
P/F: 0
C: 0
Control Opcode: 3 Unnumbered Information
PID: 0xF0 No L3

AX.25 Data:

```
86 a2 40 40 40 40 60 96 96 68 90 8a 94 7f 03 f0
```

IL2P Data Prior to Scrambling and RS Encoding:

```
63 f1 40 40 40 00 6b 2b 54 28 25 2a 0f
```

IL2P Data After Scrambling and RS Encoding:

```
6a ea 9c c2 01 11 fc 14 1f da 6e f2 53 91 bd
```

AX.25 I-Frame

This is an AX.25 I-Frame with 9 bytes of information after the 16 byte header.
Destination Callsign: `KA2DEW-2`
Source Callsign: `KK4HEJ-2`
P/F: 1
C: 1
N(R): 5
N(S) 4
AX.25 PID: 0xCF TheNET
IL2P Payload Byte Count: 9

AX.25 Data:

```
96 82 64 88 8a ae e4 96 96 68 90 8a 94 65 b8 cf 30 31 32 33 34 35 36 37 38
```

IL2P Scrambled and Encoded Data:

```
26 13 6d 02 8c fe fb e8 aa 94 2d 6a 34 43 35 3c 69 9f 0c 75 5a 38 a1 7f f3 fc
```

References for Further Study

General background on Polynomial Codes, Error Detection, and Error Correction:
Widjaja, Indra and Leon-Garcia, Alberto. *Communication Networks.* New York: McGraw-Hill 2004 166-190. Print.

A good primer on Reed Solomon codes from the BBC:
https://downloads.bbc.co.uk/rd/pubs/whp/whp-pdf-files/WHP031.pdf

James Miller's G3RUH 9600 Modem:
https://www.amsat.org/amsat/articles/g3ruh/109.html

Another 9600 modem implementation by John Magliacane KD2BD:
https://www.amsat.org/amsat/articles/kd2bd/9k6modem/9k6modem.html

The AX.25 2.2 specification:
http://www.ax25.net/AX25.2.2-Jul%2098-2.pdf

The FX.25 draft specification:
http://www.stensat.org/docs/FX-25_01_06.pdf

Wikipedia DEC SIXBIT encoding:
https://en.wikipedia.org/wiki/Six-bit_character_code#DEC_six-bit_code

Wikipedia Linear Feedback Shift Registers:
https://en.wikipedia.org/wiki/Linear-feedback_shift_register

KISS Protocol
www.ax25.net/kiss.aspx

This document was written by Nino Carrillo, reachable at nino.carrillo@outlook.com.

Changes:
26 Jan 2020 v0.3: Updated dead link to AX25 specification.
1 Aug 2020 v0.4: Added Max FEC scheme (16 parity bytes per block), updated protocol efficiency graph.