TrailZYB

John Pertell
Undergraduate Computer Science Capstone
Capstone Advisor: Dr. Michael J. Reale
SUNY Polytechnic Institute
Spring 2025

Introduction

TrailZYB is a software solution to capture and transmit landscape images from remote areas with no cellular coverage. Utilizing amateur radio frequencies allocated by the FCC, TrailZYB provides software utilities that encodes, transmits, decodes, and displays images alongside any other metadata the user may want to provide. Operating TrailZYB requires an FCC amateur radio license which is subject to FCC Part 97 rules & regulations.

TRAILzyb operates atop the KISS[3] and AX.25[2] protocol stack, ensuring compatibility with common software and hardware Terminal Node Controllers (TNCs). These protocols also provide built-in error detection through checksums.

TRAILzyb introduces a customized encoding approach that enhances packet efficiency by using specific delimiters to mark the start and end of image frames. A core technical challenge lies in the bandwidth limitations of amateur radio. Under ideal conditions, the 2-meter band allows for a maximum throughput of 1200 baud — or roughly 150 bytes per second. While this may seem sufficient, transmitting a 500x500 7-bit image could take nearly 30 minutes.

TRAILzyb addresses this constraint with a custom image encoding scheme optimized for speed and reliability. It also implements mechanisms to preserve image integrity in the event of partial data loss during transmission.

Related work

Amateur radio image transmission has been explored extensively in both analog and digital domains. One traditional method is Slow-Scan Television (SSTV)[4], a technique developed in the mid-20th century to transmit still images over narrowband radio channels. SSTV[4] encodes color images into analog audio tones that can be transmitted and decoded into viewable images at the receiver. Although SSTV[4] is simple and requires minimal bandwidth, its analog nature makes it susceptible to noise and distortion during transmission, often resulting in significant image degradation.

Modern digital techniques, such as those used in Mars rover communications [5], provide an interesting contrast. NASA rovers like Curiosity use highly robust digital encoding, error correction, and relay through satellites like the Mars Reconnaissance Orbiter (MRO)[5] to transmit images back to Earth. These systems prioritize reliability over speed, often accepting long delays to guarantee data integrity.

Although the bandwidth available to a Mars rover is significantly higher than amateur radio allocations, the underlying problem, transmitting critical image data over constrained, unreliable links is very similar.

Compared to SSTV's purely analog approach, TrailZYB employs a hybrid digital strategy. Images are compressed into custom 7-bit encodings that minimize transmission time, then structured into digital packets using the AX.25[2] and KISS[3] protocols. This preserves image integrity while allowing for partial reconstruction even under packet loss, something traditional SSTV cannot easily recover from.

Digital Amateur Radio Packet (ARDOP)[6] modes and experimental protocols like VARA[7] also attempt to push more data across narrow amateur radio bands by improving modulation and compression techniques. However, these protocols often require proprietary software or licensing. TrailZYB was specifically designed to avoid such limitations, using only open standards and custom lightweight encoding methods.

Method

1. System Overview

The TRAILzyB system is designed to enable remote image capture and transmission over amateur radio networks where no cellular or internet infrastructure is available. The system operates through a sequence of coordinated components that automate image capture, encoding, transmission, reception, and decoding.

The general operation begins with a camera connected to either a standard computer or a low-powered mobile station. The mobile station is typically based on a microcontroller platform, such as a Raspberry Pi or similar device, which is triggered by an external controller, like an ESP32. When triggered, the mobile station captures an image using an attached webcam. The image is then processed, cropped to a standardized size, optionally sharpened, and saved locally as a PNG file.

Immediately after capturing, the image undergoes encoding into a custom .zyb format. This encoding process reduces each pixel's color data into a compact 7-bit value to optimize transmission speed and resilience to packet loss. The resulting .zyb file represents the entire image as a simple sequence of 7-bit pixel encodings.

The encoded image is then framed into AX.25[2] UI packets and further encapsulated in KISS[3] protocol frames to ensure compatibility with amateur radio TNCs such as Direwolf[1]. These frames are

transmitted using TCP/IP sockets to the TNC software, which modulates and sends the data over radio frequencies.

On the receiving end, another station or server listens for incoming KISS[3] frames. Once received, the frames are decoded, stripping away the KISS [3] and AX.25[2] headers to recover the .zyb data. The .zyb file is then decoded back into a reconstructed image. Thanks to the row-based transmission structure, partial image loss results only in missing rows, allowing even incomplete transmissions to yield usable visual information.

2. Image Capturing

Image capturing is a key part in the TRAILzyb project. Image capturing is the first step in the transmission process. Images can be captured in one of two ways. Through a regular machine or through a “mobile station”. A mobile station could be a simple microcontroller (tested using raspberry pi) that has access to a web camera and UNIX-like operating system.

A regular machine would have this process automated through a provided “cap.py” script located in the “zyb_txrx/cap_image/cap.py” directory. This script executes the main functions of four other scripts in sequential order. This order is, “cap_image.py”, “encrypt.py”, “decrypt.py” and “rescaler.py”. All in their respective order. The “cap-image.py” is the heart of image capturing and therefore will be discussed in this section. Encrypt & Decrypt will be discussed in section 3, Image Processing.

The mobile station’s process for capturing images is very simple. The station uses the same capture image scripts discussed previously, but for automation it relies on a separate install. A shell script located in the root directory of the project maintains this separate install. This install sets up a separate daemon process on the device. This daemon runs from the “zyb-fireGPIO” also located in the root directory. This script works by checking for a voltage on input pin, 17. When a voltage signal is read, the script tells the operating system to open two new threads. One thread to launch the software TNC, the other to capture & transmit the image from cap_image.py. This process allows for another low-powered device to control when transmission can be completed.

After capturing an image, the code crops it to a specified size centered within the original image. If sharpening is enabled, a sharpening filter is applied to enhance the image details. Finally, the cropped (and possibly sharpened) frame is saved with a timestamped filename in PNG format.

2.1 ESP Power-Saving Image Capturing Process

In consideration of power consumption, I sought after a software solution to only control when pictures were taken. I chose two times a day to capture and transmit an image.

The low-powered device used in this project was the ESP-32. This was chosen because of the power consumption and size of the device. Code for the signal processing is in the “esp” directory of the project. The ESP-32 uses a state machine to understand when to power the raspberry pi and when to capture an image. See **Figure 2.1.1 for the state machine diagram.**

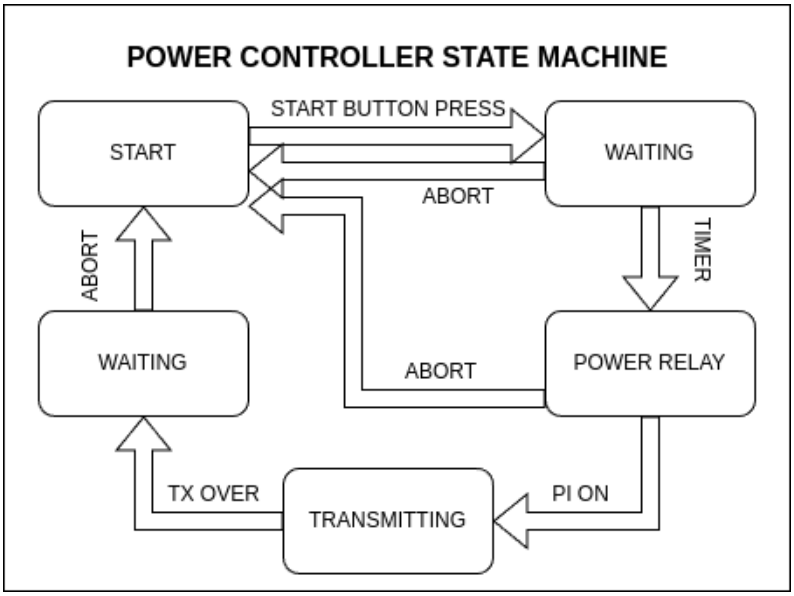

FIGURE 2.1.1 — A SIMPLE STATE MACHINE OUTLINING THE CONTROL FLOW FOR THE INTERNAL POWER CONTROLLER SOFTWARE.

3. IMAGE PROCESSING

Captured images are stored as png images and encoded as zyb images. The transmission software is designed specifically to handle zyb images. A ZYB image is useful because each row in the picture can be transmitted as one buffer. By dividing an image into rows, a lost transmission packet only results in a single lost row. Even after a handful of rows are lost, a picture can still be reconstructed with little known loss. See **Results** for a visual representation of image reconstruction.

3.1 IMAGE ENCODING

Images are encoded in the “encrypt.py” script. This script is also run automatically by “cap.py” after an image is captured. This script knows which image to encrypt by using the most recent timestamped image. Recall the image capturing system saves image names according to their timestamps.

The encryption process for generating .zyb files involves converting each pixel of a captured image into a compact 7-bit value. Each pixel in a typical color image is composed of three 8-bit components: red,

green, and blue (RGB). To reduce this information while preserving a basic representation of color, the encryption method compresses these 24 bits down to just 7 bits per pixel. **See Figure 3.1.1** for a visual representation of all the colors on the ZYB color palette.

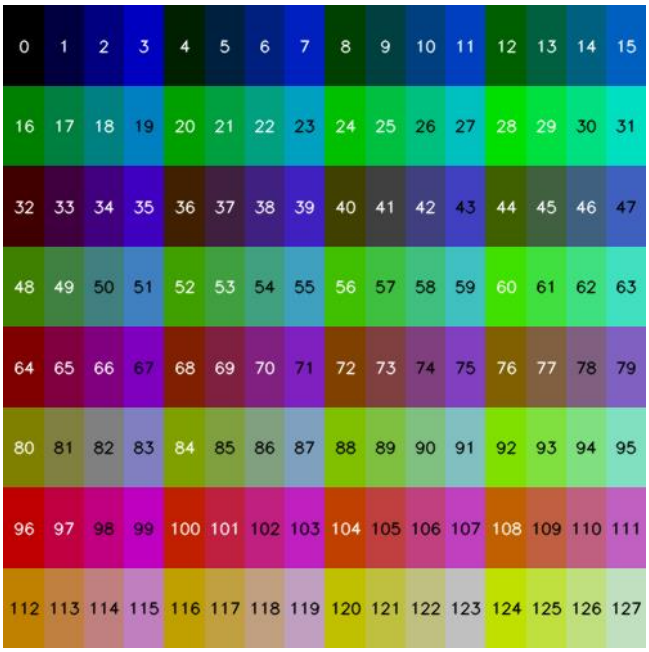

FIGURE 3.1.1 — THE 128 COLOR ZYB ENCODING COLOR PALETTE

For each pixel, the red value is scaled down to 2 bits by integer division by 64 (r // 64), limiting it to values between 0 and 3. The green component, which often carries more luminance information for the human eye, is reduced to 3 bits by dividing by 32 (g // 32), allowing values between 0 and 7. The blue component, like red, is also reduced to 2 bits (b // 64).

After scaling, these reduced values are combined into a single 7-bit number: the red 2 bits are shifted left by 5, the green 3 bits are shifted left by 2, and the blue 2 bits are added directly. A 1 is logically OR'ed at the end to ensure that the least significant bit is always set. The least significant bit's value always being a 1, indicates that a byte can never be zero, but it could have no color(black). This way the lowest value transmitted will always be a 1 and never nothing. **See Figure 3.1.2** for a visual representation of each byte looks like.

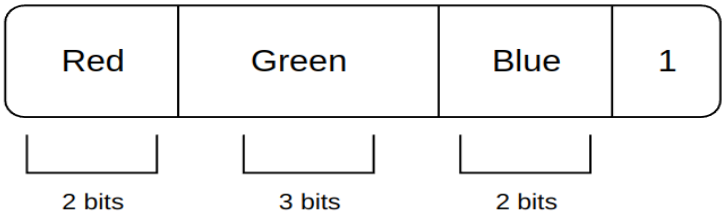

FIGURE 3.1.2 — THE ZYB PIXEL'S ENCODING SCHEME

This approach effectively encodes a pixel's essential color characteristics into a much smaller space, heavily compressing the original image's data. Once encoded, each 7-bit value is written sequentially to a text file, resulting in a simple but compact representation of the entire image in .zyb format.

3.2 IMAGE DECODING

The decoding of .zyb files is handled by the decrypt.py script. This script reads the compressed image data stored in the .zyb file and reconstructs an approximate version of the original image. The input file is expected to contain a sequence of integers, each representing a single pixel's encoded 7-bit color information.

Upon reading the file, each line is validated to ensure it represents an integer within the allowable 7-bit range. Any invalid or out-of-range values trigger a warning, and the reading process truncates accordingly to maintain data integrity. The script also ensures that the total number of pixels forms complete rows based on a fixed image width of 250 pixels; if necessary, excess pixels are discarded.

Each 7-bit value is decoded by extracting the individual color components: the red component is obtained from the two most significant bits, the green component from the following three bits, and the blue component from the two least significant bits. These extracted values are then expanded back to approximate their original 8-bit scale: red and blue are scaled by a factor of 64, while green is scaled by a factor of 32. This scaling restores each color channel to a valid 8-bit range.

The decoded pixel values are assembled into a two-dimensional image array, where each pixel's blue, green, and red values are assigned to their respective channels. Finally, the reconstructed image is saved in .png format.

4. AX.25/KISS FRAME PROCESSING

In this project, frame processing refers to the encoding and decoding of image data into a frame format suitable for transmission over a radio network using the AX.25[2] and KISS[3] protocols. These frames

are built to be compatible with the Direwolf[1] software TNC, which acts as a modem for packet radio communication.

4.1 AX.25/KISS Frame Encoding

Frame encoding is handled primarily within kisszyb.c. The image data, after being compressed and converted into a .zyb format, is prepared for transmission by first being placed into an AX.25 UI (Unnumbered Information) frame. Each AX.25[2] frame contains a destination callsign, a source callsign, control and protocol identification fields, and the information payload (i.e., the encoded image data).

To create a valid AX.25[2] frame, each callsign is encoded by left-shifting each character by one bit and appending a special flag in the last byte to indicate whether the address field is the final one in the address list. Once the addresses are encoded, the control and PID fields are appended, followed by the information data. The full AX.25[2] packet is then passed through a KISS[3] encoder, which escapes any special control characters to ensure that they are not misinterpreted during transmission.

The encoding of each 7-byte address field follows the formula:

$$\text{Encoded Address Byte} = (\text{ASCII character} \ll 1)$$

where ≪ denotes a bitwise left shift by one bit. The seventh byte (SSID and flags) is calculated as:

$$\text{SSID Byte} = ((\text{SSID}\&0x0F) \ll 1) \mid 0x60 \mid (\text{Final Flag})$$

where:

- SSID is the 4-bit substation ID
- 0x60 sets the reserved bits per AX.25[2] standard
- Final Flag is set to 0x01 if this address is the final address field.

4.2 AX.25/KISS Frame Decoding

Frame decoding follows the inverse process. Upon receiving a frame from the Direwolf[1] TNC, the raw data is extracted by removing the AX.25[2] header fields and focusing on the information section. Any KISS-specific escape sequences are reversed to restore the original data bytes. After removing the header, the payload consists solely of the encoded .zyb data, which can then be processed for further reconstruction into an image.

5. Transmission Layer

The transmission layer of the system is responsible for sending and receiving data over a network connection to the Direwolf[1] TNC, which ultimately handles RF transmission and reception. Socket programming, using TCP/IP, is utilized for this purpose and is implemented in the kisszyb.c module.

When sending a frame, a TCP socket connection is established to Direwolf's [1]KISS TCP port. The previously built KISS-encoded frame is transmitted over this socket. Proper error checking ensures that socket creation, connection, and data transmission are all handled reliably. After sending, the socket is cleanly closed.

Receiving frames works in a similar manner: a TCP socket connects to the Direwolf[1] server, listens for incoming data, and reads it into a buffer. After receiving the data, the decode process is triggered to strip away AX.25[2] and KISS[3] formatting, leaving behind only the useful payload.

Results

While testing TRAILzyB in a field environment, I deployed the mobile station, a Raspberry Pi running the TRAILzyB software stack, powered via a rechargable battery system. An ESP32-based controller was connected to monitor GPIO pin 17, configured to send a signal twice per day based on an internal state machine. At 7:00 AM, the ESP32 raised the signal pin, and I observed the system come to life.

The Raspberry Pi, upon receiving the trigger, initialized Direwolf as the software TNC and launched the image capture routine. Within a few seconds, the onboard webcam captured a still image of the landscape. The image was immediately encoded into the .zyb format, compressed using the 7-bit pixel encoding described in Section 3.1. The .zyb file was then framed into AX.25 UI packets and KISS-encoded before being handed off to Direwolf for RF transmission.

On the receiving end, located approximately 3/4 of a mile away using an omni-directional antenna, I had a second machine running Direwolf in receive mode. The encoded data was successfully received and written to disk. TRAILzyB's decoding script parsed the .zyb file, reconstructed the image, and saved the result. The image was displayed successfully, with no packet loss. **Figure R.1** shows the image received.

FIGURE R.1 — FIELD TESTED IMAGE RECEIVED FROM MOBILE STATION

The following figures illustrate TRAILzyB's performance under varying packet loss conditions. The original image comes from the field-tested image, except packet loss was simulated at the receiver to test the impact on reconstruction quality. Even with substantial data loss, key visual features of the scene remain detectable.

- **Figure R.2 (2nd photo)**: Reconstructed image with ~25% packet loss
  - 25% packet loss equates to ~60 lost frames.
- **Figure R.2 (3rd photo)**: Reconstructed image with ~50% packet loss
  - 50% packet loss equates out to ~125 lost frames

FIGURE R.2 — FIELD TESTED IMAGE WITH SIMULATED RANDOMIZED PACKET LOSS. LOSSES ARE 0% LOSS(ORIGINAL), ~25% LOSS, ~50% LOSS, LEFT-TO-RIGHT, RESPECTIVELY.

In TRAILzyB, each image is transmitted row-by-row, with each row requiring approximately 3 seconds to send over a 1200 baud AX.25 link. Because this timing is consistent and repeatable, total transmission time is directly proportional to the number of image rows. For a square image, the total transmission time in minutes can be expressed as:

$$T=\frac{\tau \cdot N}{60}$$

Where:

- T = total transmission time in minutes
- N = number of image rows
- τ = time per row in seconds
- τ = 3 seconds for stable TRAILzyB transmission (default)

Image Size (pixels)	Rows Sent	Total Time (seconds)	Total Time (minutes)
250 × 250	250	750	12.5
500 × 500	500	1500	25.0
1000 × 1000	1000	3000	50.0

TABLE 1 — TRANSMISSION DURATIONS FOR SQUARE IMAGE RESOLUTIONS USING TRAILZYB.

This predictable behavior allows users to make informed tradeoffs between image quality and transmission time. Reducing image height directly diminishes the total transmission time in a linear and controllable way. Best performance was achieved with an original image size of 250 x 250 pixels.

Discussion

A major success of this project was the ability to reliably transmit and receive ZYB-encoded images over a simulated amateur radio environment. The core encoding, transmission, and decoding pipeline worked consistently, allowing images to be reconstructed even with occasional packet loss. The design of the 7-bit encoding format proved to be both compact and resilient during testing.

However, a significant challenge encountered during development was debugging long transmission processes. Because each image transmission took several minutes, diagnosing problems and iterating on fixes was time-consuming. Small changes to the encoding or transmission logic often required full test

cycles, delaying development. This highlighted the importance of designing more efficient debugging tools or simulation environments for projects involving low-bandwidth communications.

Additionally, developing the mobile station controller using FreeRTOS on the ESP32 taught important lessons about real-time task scheduling and GPIO management. Initial versions of the mobile station had difficulty handling GPIO interrupts reliably, occasionally missing button presses or firing incorrectly. These issues were resolved through improved debouncing logic and restructuring the interrupt handling flow.

If I were to redesign the system, I would focus on improving the mobile station's power management approach. Instead of relying solely on timed relays and manual control, I would integrate a fully autonomous, solar-powered controller. Such a design would allow the mobile station to operate indefinitely in remote environments, automatically managing its own power and transmission schedule based on available solar energy.

Conclusion and Future Work

This project demonstrated the design and implementation of TRAILzyB, a complete software system for capturing and transmitting landscape images from remote areas using amateur radio frequencies. The system successfully integrates image capture, custom 7-bit encoding, AX.25 and KISS packet framing, and reliable end-to-end transmission through the Direwolf software TNC. The row-based encoding format provides a simple yet effective way to reduce transmission time and mitigate the impact of packet loss, enabling partial image reconstruction even under degraded conditions.

If this project were to be extended further, one valuable improvement would be to develop a centralized system that actively receives image data and automatically uploads it to a publicly accessible web interface. This would allow users to monitor and archive images from remote stations in real time, creating an open, distributed image feed sourced almost entirely via amateur radio. Such a platform could integrate cloud storage, geotagging, and user-submitted stations, ultimately enabling a community-driven network for low-bandwidth environmental monitoring.

References

[1] "Dire Wolf: A Software 'Soundcard' TNC and APRS Encoder/Decoder," GitHub, Available: https://github.com/wb2osz/direwolf. [Accessed: Apr. 28, 2025].

[2] "AX.25 Amateur Packet-Radio Link-Layer Protocol," TAPR, Available: https://www.tapr.org/pdf/AX25.2.2.pdf. [Accessed: Apr. 28, 2025].

[3] Ph. Karn, "The KISS TNC: A simple host-to-TNC communications protocol," AMRAD Newsletter, Jan. 1987. Available: https://www.ka9q.net/papers/kiss.html. [Accessed: Apr. 28, 2025].

[4] "Slow Scan Television (SSTV)," American Radio Relay League (ARRL). Available: https://www.arrl.org/slow-scan-television. [Accessed: Apr. 28, 2025].

[5] NASA/JPL-Caltech, "Curiosity Rover Communications: How the Mars Science Laboratory Rover Communicates," NASA Mars Exploration Program. Available: https://mars.nasa.gov/msl/mission/communications/. [Accessed: Apr. 28, 2025].

[6] "Amateur Radio Digital Open Protocol (ARDOP)," Winlink Development Team. Available: https://www.winlink.org/ARDOP. [Accessed: Apr. 28, 2025].

[7] "VARA HF Modem," EA5HVK Official Site. Available: https://rosmodem.wordpress.com/vara/. [Accessed: Apr. 28, 2025].

Appendix

TRAILzyB has been tested and confirmed to work on Raspberry Pi OS version 6.6 for mobile stations and Ubuntu version 24.04.1 for general-purpose desktop stations. It relies on several key software dependencies, including Python 3 (tested with Python 3.11 or later), OpenCV for Python (opencv-python), the Direwolf software Terminal Node Controller (TNC) for handling AX.25 and KISS framing, GNU Make for building native C modules.

Before using TRAILzyB, it is necessary to ensure that Direwolf is properly installed and configured. Direwolf serves as the software modem that translates KISS-framed packets into audio tones suitable for amateur radio transmission. Detailed instructions for Direwolf installation and setup can be found at the Direwolf GitHub Repository, https://github.com/wb2osz/direwolf.

The source code and installation scripts for TRAILzyB are available at: https://github.com/Johnny4251/TRAILzyb.

To install the system on a general-purpose station, users should clone the repository, make the installation script executable, and run it with administrative privileges. The installation script, install.sh, updates the system's package index, installs necessary dependencies such as python3-venv, and direwolf, sets up a Python virtual environment, installs the required Python packages, builds and installs the kisszyb C module, and prepares all transmission scripts and binaries for use.

For mobile station deployment, a similar process is followed using a separate installation script, install-mobile.sh, designed for lightweight Raspberry Pi setups. This configuration is intended for automated, low-power operation triggered by external GPIO inputs, enabling flexible and remote deployment scenarios.