piQSL

Draft Technical Specification

piQSL

piQSL *(noun)*
Pronounced: **pixel**

1. A novel digital radio mode designed to gamify amateur radio communication by transmitting visually engaging pixel art in place of traditional QSL card exchanges.
2. Transmissions begin with a header containing the sender's and recipient's callsigns, followed by a 32x32 pixel art image. The mode emphasizes creativity, collectability, and the spirit of radio communication.

Introduction

piQSL is a digital radio mode that is being developed to encode, transmit, and decode 32x32 grid image and meta data over, predominantly, HF bands. In truth, the design is band and modulation agnostic and can theoretically be transmitted over any voice mode. However, with the narrowband transmission, it is best suited for HF bands where there is often good support on most radios for narrowing the bandwidth, and of course, it is the home of DX'ing.

The motivation to create this mode is really down to a personal frustration with amateur radio. I have an affinity with radio, I think it's in my blood; my Grandfather, also an engineer, received an MBE for rebuilding an Enigma relay transmitter in the Netherlands during WW2 by stripping electrical components and rebuilding the transmitter after bombing. Crazy. This story has always resonated with me and made radio seem somewhat whimsical. However, I'm not much of a talker, and there is only so much listening you can do, and other digital modes haven't really excited me. SO, I have found myself with a license, and never using it.

piQSL aims to bring a new type entertainment and excitement into radio with a specific focus on collecting digital 32 x 32 QSL type cards through your contacts directly. Gamifying the process making contacts, and bringing motivation to myself to participate actively in digital modes where I can start making contacts without talking, or getting bored!

This document provides an in-depth technical overview of the encoding, modulation, and decoding methodologies that will be used in the piQSL digital radio mode.

Digital Modulation

Modulation

Modulation is obviously one of the most important aspects of a radio mode. MFSK (Multi-Frequency Shifted Keys) had already been decided on, for it's ease of encoding and it's a proven method, but the final specification is dependent on testing and having the freedom to tune the modulation quickly and easily to find the most effective combination.

In order to process the data that we want to be able to transmit, a non-standard MFSK modulation is required. Having researched and tested the hardware limitations, and tested multiple options regarding bandwidth, tone duration, tone frequencies etc the following modulation scheme has been decided upon for testing; a **final specification** will be confirmed after testing is completed:

Frequency Range and Bandwidth

In order to fine tune the modulation specification, the modulation can be tuned rapidly for testing purposes with variable within `tuning.js`:

- **Minimum Tone Frequency (`MIN_TONE_FREQ`):**
  This is the lowest frequency used for transmitting data tones.
- **Bandwidth (`BANDWIDTH`):**
  The total frequency range for data transmission.
- **Reserved Calibration Bandwidth:**
  A portion of the bandwidth is reserved for calibration tones at both ends of the frequency range.

  ```
  const reservedCalibrationBandwidth = BANDWIDTH / 7;
  ```

- **Available Bandwidth for Data:**
  The usable bandwidth for data tones is the total bandwidth minus the reserved calibration bandwidth.

```
const availableBandwidth = BANDWIDTH - reservedCalibrationBandwidth;
```

Tone Mapping

For simplicity and transmission robustness, the `stepSize` (the frequency gap between tones) is fixed across the tone maps, based on the largest tone map in the schema.

Step Size Calculation

- **Step Size (`stepSize`):**
  The frequency difference between adjacent tones is calculated based on the available bandwidth and the number of tones needed.

  ```
  const stepSize = availableBandwidth / 39;
  ```

  - **Why 39?**
    - 38 characters (26 letters A-Z, 10 digits 0-9, hyphen '-', and space ' ')
    - Plus one special **End-of-Line (EOL)** tone

Character Frequency Map

Character Frequency Map(`CHAR_FREQ_MAP`)

- **Characters Mapped:**
  'A' to 'Z', '0' to '9', '-', and ' ' (space)
- **Frequency Assignment:**
  Each character is assigned a unique frequency starting from the minimum tone frequency, incremented by the step size.

  ```
  for (let i = 0; i < characters.length; i++) {
      charFrequencyMap[characters[i]] = Math.round(MIN_TONE_FREQ + (i * stepSize));
  }
  ```

- **End-of-Line (EOL) Tone:**
  A special tone used to signify the end of a line or data block.

```
charFrequencyMap['EOL'] = Math.round(MIN_TONE_FREQ + (characters.length * stepSize));
```

Color Tone Maps

- **32-Color Tone Map (`_32C_TONE_MAP`):**
  Used for image data where each of 32 colors is mapped to a unique frequency.

```
for (let i = 0; i < 32; i++) {
    tone32CMap.push(Math.round(MIN_TONE_FREQ + (i * stepSize)));
}
```

- **4-Tone Map (`_4T_TONE_MAP`):**
  Used for a reduced color mode or specific data types, selecting every 8th tone from the 32-tone map.

```
for (let i = 0; i < 4; i++) {
    tone4TMap.push(Math.round(tone32CMap[i * 8]));
}
```

Calibration Tones

- **Calibration Tones:**
  Special tones at the minimum and maximum frequencies reserved for synchronization and calibration purposes.

```
const CALIBRATION_TONE_MIN = toneMaps.MIN_TONE_FREQ;
const CALIBRATION_TONE_MAX = toneMaps.MAX_TONE_FREQ;
```

Transmission Details

Transmission is via audio. Support should be available for CAT, but **piQSL** will be able to function perfectly well by direct connection to audio feeds; line-in or mic for receiving, and line-out or phones for transmission; just remember to PTT!

Tone Duration

- **Standard Tone Duration (`TONE_DURATION`):** in milliseconds* per tone
- **Header Tone Duration (`HEADER_TONE_DURATION`):** in milliseconds* per tone for header data
- **Calibration Tone Duration :** Fixed to 500ms for each tone.

'*' Technically limited to a minimum of approximately 30ms, practically limited to a minimum of about 50ms

Transmission Sequence

1. **Calibration Tones:**
   Transmission starts with calibration tones at the minimum and maximum frequencies to help the receiver calibrate frequency drift and synchronisation.

   ```
   await changeTone(CALIBRATION_TONE_MIN, 500);
   await changeTone(CALIBRATION_TONE_MAX, 500);
   ```

2. **Header Transmission:**
   The header contains metadata; sender callsign, recipient callsign, or CQ, and mode (e.g., '4T' or '32C'). Each character in the header is transmitted using its corresponding frequency from the `CHAR_FREQ_MAP`. The header is limited to 15 characters, and buffered to 15 if the transmitted data is less than 15. This can be expanded upon in later variants.

   ```
   const headerString = `${senderCallsign}-${recipientCallsign}-${mode}`.padEnd(15, '
   ');
   ```

3. **Data Transmission:**
   - **Image Data:**
     Image pixels are mapped to color indices, which are then mapped to frequencies using either the `_32C_TONE_MAP` or `_4T_TONE_MAP`, depending on the mode.
   - **Tone Sequence:**
     Tones are transmitted sequentially, with calibration tones inserted between them for synchronisation of decoding.

     ```
     for (let i = 0; i < tones.length; i++) {
         // Insert calibration tone between characters
         await changeTone(CALIBRATION_TONE_MIN, TONE_DURATION);
     ```

```
    // Transmit data tone
    await changeTone(tones[i], TONE_DURATION);
}
```

4. **End-of-Line Tone:**
   An `EOL` tone is transmitted to signify the end of a line or data block.

   ```
   await changeTone(END_OF_LINE, TONE_DURATION * 2);
   ```

Smooth Transitions

- **Frequency Transitions:**
  The modulation uses smooth transitions between tones to reduce abrupt frequency changes, which can help minimise spectral splatter and improve signal quality; hard frequency transitions can cause many audio glitches, especially on lower quality devices.

  ```
  if (USE_SMOOTH_TRANSITIONS) {
      oscillator.frequency.linearRampToValueAtTime(frequency,
  txAudioContext.currentTime + 0.02);
  } else {
      oscillator.frequency.setValueAtTime(frequency, txAudioContext.currentTime);
  }
  ```

Demodulation Details

FFT Analysis

- **FFT Size (`FFT_SIZE`):** 2048
  Determines the frequency resolution of the Fast Fourier Transform (FFT) used in the receiver.
- **Time and Frequency Resolution:**
  Larger FFT sizes provide better frequency resolution but require longer time durations per tone (further details on this in the Interpolation Methods section).

- **Goertzel and Hamming Window:**
  Hamming Window will prepare our signal for analysis by smoothing the edges, reducing artefacts and marginally improving on the frequency resolution. Then Goertzel will determine which frequencies from the modulation scheme are most powerful in the sample.

Demodulation Parameters

- **Amplitude Threshold (`RX_AMPLITUDE_THRESHOLD`):** -50 dB (default)
  Signals below this amplitude are ignored to reduce noise. This is currently coded in the `tuning.js` file, however, it will be adjustable in the UI using the waterfall to visually hide the noise floor.
- **Analysis Interval (`RX_ANALYSIS_INTERVAL`):** in milliseconds
  Determines how often the receiver samples the incoming signal. The sampling rate *can* overlap the time resolution allowing for multiple samples to be taken over various point within a single time resolution, there is a hard limit on this figure based on the processor speed of the computer being used - this will be the limiting factor in low specification SBC's i.e. Raspberry Pi; lower 10ms is unlikely on RPi - tone duration will ensure that low low specification devices are able to demodulate effectively.
- **Required Samples per Tone (`RX_REQUIRED_SAMPLES_PER_TONE`):** 5
  The number of consecutive samples required to confirm the detection of a tone.
- **Calibration Drift (`RX_CALIBRATION_DRIFT`):**
  Allows for frequency drift in calibration tones due to clock inaccuracies or Doppler shifts.

  ```
  const RX_CALIBRATION_DRIFT = BANDWIDTH / 7; // Approximately 142.86 Hz
  ```

Synchronization

Demodulation is quite robust to synchronisation drift, due to the two-tone calibration transmission, the demodulation functions are triggered by the demodulation of the second calibration tone, which

can be received any time from the 6th second, up to the 15th second of the scheduled period.

Transmission Scheduling

- **Processing Interval (`PROCESSING_INTERVAL`):**
  Transmissions are scheduled to start on the 7th second of every x minutes, where x is determined by the below calculation of the transmission duration;

```
const PROCESSING_INTERVAL = Math.ceil(((TONE_DURATION * 2 * 1024) +
(HEADER_TONE_DURATION * 2 * 15) + 15000) / (1000 * 60));
```

- **Start Time (`RX_startTime`):**
  Receiver starts listening at a specific time offset after the minute mark (e.g., +6 seconds).
- **End Time (`RX_endTime`):**
  Receiver times out if no calibration tone is detected by a certain time (e.g., +15 seconds).

Countdown Logic

- The application calculates the time until the next synchronised transmission interval and schedules the transmission accordingly.

```
function scheduleTransmission(gridData, senderCallsign, recipientCallsign, mode) {
    // Calculate next transmission time
    const nextInterval = new Date(epoch.getTime() + Math.ceil(timeSinceEpoch /
intervalMs) * intervalMs);
    nextInterval.setUTCSeconds(7); // Set seconds to +7 as required
    // Schedule transmission
}
```

Summary of Scheme

- **Modulation Scheme:** Multi-Frequency Shift Keying (MFSK)
- **Data Encoding:** Each symbol (character or colour) is mapped to a unique frequency within the specified bandwidth.

- **Calibration:** Special tones at reserved frequencies help synchronise the transmitter and receiver, accounting for any frequency drift.
- **Transmission Flow:**
  1. Send calibration tones.
  2. Transmit header data encoded as frequencies.
  3. Transmit image or text data encoded as frequencies.
  4. Use EOL tones and calibration tones for synchronisation between data blocks.
- **Reception Flow:**
  1. Listen for calibration tones to calibrate frequency offsets and trigger data demodulation.
  2. Use Goertzel and Hamming Wave to detect frequencies in the incoming signal.
  3. Map detected frequencies back to symbols based on the tone maps.
  4. Render the Header to the UI.
  5. Sequentially decode the Image data to the UI.
- **Synchronization:** Transmissions are scheduled at precise intervals to ensure that receivers are listening at the correct times, however drift is supported by robust demodulation.

Considerations

- **Frequency Resolution vs. Time Resolution:**
  There's a trade-off between frequency resolution and time resolution in FFT analysis. Higher `FFT_SIZE` improves frequency resolution but requires longer tones.
- **Signal Quality:**
  Using smooth transitions between frequencies and appropriate amplitude thresholds helps maintain signal integrity and reduces noise. It does however increase the error count between transitions and shortens the window of a tone.
- **Clock Synchronisation:**
  Both transmitter and receiver *should* have their clocks synchronised to the same time reference to ensure accurate timing, certainly to within a few seconds of each other.

- **Environmental Factors:**
  Factors like Doppler shifts, hardware inaccuracies, and noise can affect the transmission and reception of tones. Calibration tones and drift allowances help mitigate these issues.

Calculation Methods

Summary

piQSL will use Goertzel and Hamming Window calculations for effective frequency demodulation, due to it's speed and low computational requirements compared with the alternatives. While FFT and Interpolation does allow for larger frequency shifts, as it collects data from all frequency bins, the effect this has on performance is not an acceptable trade off.

Details of the options I looked into are below.

FFT Size

1 What is FFT Size?

Good Question, before starting this, I knew little about it myself!

FFT size is the number of samples that are processed at one time by the **Fast Fourier Transform (FFT)** algorithm. The FFT is a mathematical method used to transform a live signal into snapshots of data; like sheet music. This allows us to see the different frequencies (like musical notes) that make up the original signal. When is comes to understanding the resolution, speed and accuracy etc we can think of the FFT size a bit like the number of pieces in a puzzle, where each piece is called a **bin**:

- **Small Puzzle (Small FFT Size):** Fewer pieces mean each piece is larger. The puzzle is quicker to assemble, quicker to find the pieces, less processing power is required i.e can be done by a child) but the picture is less detailed.
- **Large Puzzle (Large FFT Size):** More pieces mean each piece is smaller. It takes longer to assemble, its takes longer to find the pieces, is harder to do, but the picture is more detailed.

2 Why do we need to know this?

For us to effectively develop the modulation, encoding and decoding of **piQSL**, we need to understand how all this affects the ability to process the signal on:

- Bandwidth requirements
- Frequency steps to keep stones clearly separated
- Minimum tone duration required to ensure signal processing is possible

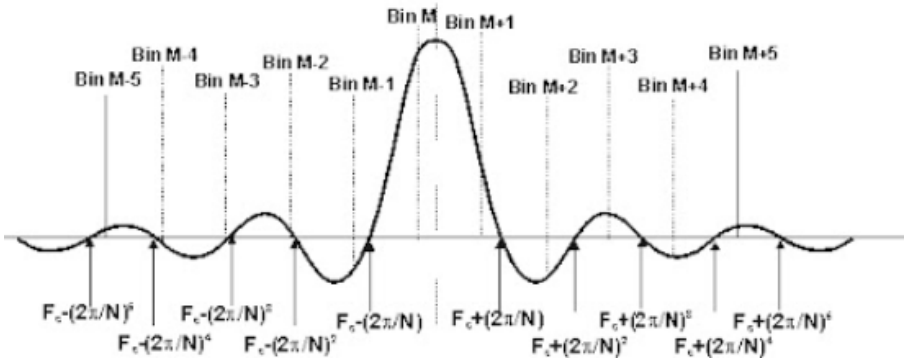
---

Interpolation Methods (Option 1)

The Basics

What is interpolation?

Interpolation is a mathematical method used to estimate unknown values that fall between known data points. In simple terms, it's a way of "filling in the gaps" in data to make a smoother or more detailed representation. For us what this means, is that we can use smaller FFT sizes, resulting in a faster time resolution, but lower frequency resolution, then use interpolation to fill in the gaps that are missing to estimate where a frequency might lie between two FFT bins:



Showing a frequency between bins, with interpolation

Methodology

Table Showing Frequency Resolutions with Different Interpolation Methods

FFT Size	Time Resolution (ms)	Frequency Resolution (Hz)	With Parabolic Interpolation (Hz)	With Quadratic Interpolation (Hz)
32	0.73	1378.13	689.06	344.53
64	1.45	689.06	344.53	172.27
128	2.90	344.53	172.27	86.13
256	5.80	172.27	86.13	43.07
512	11.61	86.13	43.07	21.53
1024	23.22	43.07	21.53	10.77
2048	46.44	21.53	10.77	5.38
4096	92.88	10.77	5.38	2.69
8192	185.76	5.38	2.69	1.34
16384	371.52	2.69	1.34	0.67
32768	743.04	1.35	0.67	0.34

Note: The "Hz" columns represent the approximate smallest frequency difference that can be resolved using each method.

Understanding the Calculations

1. Time Resolution

The time resolution (Δt) is the duration of the FFT window, for us, the longer this window is the longer the tone duration of the transmission must be. This is because for each time we trigger this the analysis of an FFT window we are trying to return a single tone i.e. the frequency with the highest amplitude for the FFT period:

$\Delta t = \frac{N}{f_s}$

- (N) is the FFT size.
- (f_s) is the sampling rate (e.g., 44.1 kHz).

Example for N = 256:

$\Delta t = \frac{256}{44100} \approx 0.00580 \text{ seconds} \approx 5.80 \text{ ms}$

2. Frequency Resolution

The frequency resolution (Δf) is the width of each frequency bin, this is the width of each bin that we are analysing per FFT window, the lower the resolution the faster we can iterate of the total number of bin per FFT window. Another speed vs. accuracy fight:

$$\Delta f = \frac{f_s}{N}$$

Example for N = 256:

$$\Delta f = \frac{44100}{256} \approx 172.27\,\text{Hz}$$

Here, our example shows that for an FFT window size of 256, we can analyse in in 5.8ms (from the time resolution), however, only to an accuracy of 172.72 Hz, this is because we've

3. Effective Frequency Resolution Without Interpolation

Without any interpolation, the smallest frequency difference we can detect is equal to the frequency resolution (Δf).

4. Parabolic and Quadratic Interpolation

Interpolation techniques improve frequency estimation by fitting a curve to the magnitude spectrum around the peak bin.

- **Parabolic Interpolation:** Assumes the peak and its immediate neighbors form a parabola.
- **Quadratic Interpolation:** A more general form that may include additional calculations for a better fit.

Improvement Factor: Interpolation methods can improve the effective frequency resolution by a certain factor. Typically:

- **Parabolic Interpolation:** Improves resolution by a factor of ~2.
- **Quadratic Interpolation:** Improves resolution by a factor of ~4.

Worked Examples

Example Using N (FFT Size) = 256

1. **Without Interpolation**
    - Frequency Resolution (Δf): **172.27 Hz**
    - Effective Frequency Resolution: **172.27 Hz**
2. **With Parabolic Interpolation**
    - Effective Frequency Resolution:

$$\text{Effective } \Delta f = \frac{\Delta f}{2} = \frac{172.27}{2} \approx 86.13 \text{ Hz}$$

3. **With Quadratic Interpolation**
    - Effective Frequency Resolution:

$$\text{Effective } \Delta f = \frac{\Delta f}{4} = \frac{172.27}{4} \approx 43.07 \text{ Hz}$$

Implementation of Interpolation Methods

All modulation tuning is controlled within the root folder file `tuning.js` there are two constants, which can be changed to switch on or off the interpolation methods used when receiving a signal which is used for testing:

```
const USE_QUADRATIC_INTERPOLATION = true; // more accurate
const USE_PARABOLIC_INTERPOLATION = false; // faster
```

All the receiving decoding in managed inside `src/receive.js` and interpolation is completed as follows:

```
if (peakIndex !== -1 && maxAmplitude >= RX_AMPLITUDE_THRESHOLD) {

        let peakFrequency;

        if (USE_QUADRATIC_INTERPOLATION) {

            // Quadratic interpolation over bins
            let mag0 = RX_dataArray[peakIndex - 1] || RX_dataArray[peakIndex];
            let mag1 = RX_dataArray[peakIndex];
            let mag2 = RX_dataArray[peakIndex + 1] || RX_dataArray[peakIndex];
            mag0 = Math.pow(10, mag0 / 20);
            mag1 = Math.pow(10, mag1 / 20);
```

```
            mag2 = Math.pow(10, mag2 / 20);
            const numerator = mag0 - mag2;
            const denominator = 2 * (mag0 - 2 * mag1 + mag2);
            const delta = denominator !== 0 ? numerator / denominator : 0;
            const interpolatedIndex = peakIndex + delta;
            peakFrequency = interpolatedIndex * binWidth;

        } else {
            // No interpolation, use the peak index directly
            peakFrequency = peakIndex * binWidth;
        }
        RX_detectTone(peakFrequency, maxAmplitude);

    }
```

Goertzel and Hamming Window (Option 2)

The Basics

Using a combination of the Hamming Window and the Goertzel Algorithm, offers a faster and more tuned extraction of the received tones, where FFT analyses the whole spectrum.

1. Hamming Window `applyHammingWondow`

The **Hamming window** is a type of window function used in digital signal processing to reduce spectral leakage when analyzing signals. When applying the Fast Fourier Transform (FFT) or similar techniques, the input signal is often truncated to a finite length. This truncation can introduce discontinuities, causing unwanted artifacts (spectral leakage) in the frequency domain.

By applying the Hamming window, we taper the signal to reduce these discontinuities, resulting in more accurate frequency analysis.

Mathematical Definition

The Hamming window is defined as:

$$w[n] = 0.54 - 0.46 \cdot \cos\left(\frac{2\pi n}{N-1}\right), \quad 0 \leq n \leq N-1$$

Hamming Window Formula

Where:

- $w[n]$: The window value at index n.
- N: The total number of samples.
- n: The current sample index (from 0 to $N-1N-1N-1$).

This creates a symmetric curve that peaks at the centre of the window.

How It Works

The input signal $x[n]$ is multiplied element-wise by the window function $w[n]$:

$$x_{\text{windowed}}[n] = x[n] \cdot w[n]$$

This operation smooths the edges of the signal, tapering it toward zero at the boundaries.

Advantages

- **Reduces Spectral Leakage**: The Hamming window ensures the transition at the edges of the signal is smooth.
- **Improves Frequency Resolution**: Frequency components become more distinguishable in the analysis.

Practical Example

Let's assume a signal $x[n]$ of length $N = 8$:

$$x[n] = [1, 0.5, 0.2, 0.1, -0.1, -0.2, -0.5, -1]$$

The corresponding Hamming window values are:

$$w[n] = [0.08, 0.31, 0.77, 1.0, 1.0, 0.77, 0.31, 0.08]$$

Applying the window:

$$x_{\text{windowed}}[n] = [1 \cdot 0.08, 0.5 \cdot 0.31, \ldots, -1 \cdot 0.08]$$

$$x_{\text{windowed}}[n] = [0.08, 0.155, 0.154, 0.1, -0.1, -0.154, -0.155, -0.08]$$

This results in a signal smoothed at the edges, ready for spectral analysis.

Implementation

```
/ Utility: Apply Hamming window to samples

function applyHammingWindow(samples) {

    const N = samples.length;

    return samples.map((sample, n) => sample * (0.54 - 0.46 * Math.cos((2 * Math.PI * n) / (N - 1))));

}
```

---

2. Goertzel Algorithm (`goertzel`)

Purpose

The **Goertzel algorithm** is a computationally efficient method for determining the power of specific frequency components in a signal. Unlike the FFT, which computes all frequency components, Goertzel focuses on individual frequencies. As we have a known set of frequency tones, this makes it an ideal candidate for demodulating **piQSL**.

Mathematical Definition

The Goertzel algorithm evaluates the Discrete Fourier Transform (DFT) for a single frequency bin. For a target frequency f_k, the DFT coefficient is calculated as:

$$X_k = \sum_{n=0}^{N-1} x[n] \cdot e^{-j\frac{2\pi kn}{N}}$$

Where:

- X_k: The DFT coefficient for frequency bin k.

- $x[n]$: The input signal at index n.
- N: The total number of samples.
- $e^{-j\frac{2\pi kn}{N}}$: The complex exponential term representing the frequency component.

LUCKILY JAVASCRIPT WILL DO THIS ALL FOR ME...I'm not even going to break this down, it's horrible, and honestly, I had to use AI create a function in JS that worked correctly as I had spent too long tyring to get it working.

Advantages

- **Efficiency**: Computes only specific frequencies, reducing computational load compared to FFT.
- **Targeted Analysis**: Ideal for detecting predefined frequencies, such as those in your modulation/demodulation process.

Implementation

```
// Utility: Goertzel algorithm for frequency detection

function goertzel(samples, sampleRate, targetFreq) {

    const N = samples.length;
    const k = Math.round((N * targetFreq) / sampleRate);
    const omega = (2.0 * Math.PI * k) / N;
    const sine = Math.sin(omega);
    const cosine = Math.cos(omega);
    const coeff = 2.0 * cosine;
    let q0 = 0;
    let q1 = 0;
    let q2 = 0;

    for (let i = 0; i < N; i++) {
        q0 = coeff * q1 - q2 + samples[i];
        q2 = q1;
        q1 = q0;
    }

    const real = q1 - q2 * cosine;
    const imag = q2 * sine;
```

```
    return Math.sqrt(real * real + imag * imag);
}
```

It looks so simple here.

Testing and Final Decision

I tested both FFT and Interpolation (parabolic and quadratic), then tested Goertzel. On average Interpolation at FFT size of 2048 performed sample computation at 4-7ms intervals. While Goertzel and Hamming Window performed at a speed of 0.3 - 1.5ms at FFT size of 8192.

In fact, in all tested metrics, Goertzel and Hamming Window outperforms FFT and Interpolation:

1. Key Data Points

Method	FFT Size	Average Computation Time
Interpolation (FFT)	2048	4-7 ms
Goertzel + Hamming	8192	0.3-1.5 ms

---

2. Speed Performance Statistics

Mean Computation Time

- **Interpolation (FFT):**
  - Average computation time: $\frac{4+7}{2} = 5.5\,\text{ms}$
- **Goertzel + Hamming:**
  - Average computation time: $\frac{0.3+1.5}{2} = 0.9\,\text{ms}$

Performance Speedup

The speedup factor is calculated as:

$$\text{Speedup Factor} = \frac{\text{Mean Time (FFT)}}{\text{Mean Time (Goertzel)}}$$

$$\text{Speedup Factor} = \frac{5.5}{0.9} \approx 6.11$$

- **Goertzel + Hamming Window** is approximately **6.1x faster** than Interpolation with FFT for processing time.

---

3. Frequency Resolution Differences

Frequency resolution is determined by the **FFT size** and the **sampling rate**It is calculated as:

$$\text{Resolution} = \frac{\text{Sampling Rate}}{\text{FFT Size}}$$

Assume a typical sampling rate of $44100\,\text{Hz}$:

- **Interpolation (FFT)**:
  - Resolution: $\frac{44100}{2048} \approx 21.53\,\text{Hz}$
  - Effective Freq: $\frac{21.53}{4} \approx 5.38\,\text{Hz}$
- **Goertzel + Hamming**:
  - Resolution: $\frac{44100}{8192} \approx 5.38\,\text{Hz}$

Comparison

- **Goertzel + Hamming Window** achieves a similar frequency resolution compared to Interpolation with FFT.

Error Correction

An important factor in the demodulation is error correction and mitigation. The easiest method to achieve better accuracy to increase the bandwidth and the tone lengths. However, this is not practical to the extent that would be required to ensure acceptable demodulation success rates. So, a number of error correction and mitigation methods have been employed.

Frequency Snapping

In order to help account for slight shifts in frequency, errors in FFT bin calculations and noise, a method for frequency snapping is built in to the demodulation functions, the snapping threshold is calculated with the following:

$$\text{Threshold} = \frac{\text{Frequency Step} \times 0.9}{2}$$

Example Calculation:

Given a Frequency Step of 10 Hz:

$$\text{Threshold} = \frac{10 \times 0.9}{2} = \frac{9}{2} = 4.5\ \text{Hz}$$

- **Example Expected Possible Tones:** 850 Hz and 860 Hz.
- **Threshold:** 4.5 Hz.
- **Frequency at 855 Hz:**
  - Distance to 850 Hz: $(|855 - 850| = 5\ \text{Hz})$
  - Distance to 860 Hz: $(|860 - 855| = 5\ \text{Hz})$
  - Since 5 Hz > 4.5 Hz (Threshold), 855 Hz is considered an error frequency and not logged.
- **Frequency at 856 Hz:**
  - Distance to 860 Hz: $(|860 - 856| = 4\ \text{Hz})$
  - Since 4 Hz ≤ 4.5 Hz (Threshold), 856 Hz snaps to 860 Hz.

Explanation:

- The **Threshold** determines the maximum allowable deviation from an expected tone frequency for it to be considered valid.
- The threshold, is **proportional** to the bandwidth and step size, allowing for simple tuning during development.

- Frequencies within the Threshold of an expected tone are "snapped" to that tone.
- Frequencies outside the Threshold are considered errors and are not logged.

Consecutive frequency sequencing and deletion

```
const RX_REQUIRED_SAMPLES_PER_TONE = 6; // how many consecutive samples of a tone required
const RX_MIN_SAMPLES_PER_TONE = 3;
```

With this integer we are using it to confirm a tone by ensuring that the tone is received in that many consecutive samples. In addition to this, any consecutive tone with an element count of < the `RX_MIN_SAMPLES_PER_TONE` is removed from the array, thus allowing the for a potentially shorter run of consecutive elements to be become longer. Then Finally, any consecutive elements of an array that meet the `RX_REQUIRED_SAMPLES_PER_TONE` get added to a new final array of received tones.

It is a simple yet elegant method of error correction by post-processing data arrays.

Example:

```
rawArray = [1,1,1,2,2,1,1,1,3,3,3,3,3,3,3,3,1,2,1,1,1,2,2,1,1,1,4,4,4,4,4,4,];

dropMin = [1,1,1,1,1,1,3,3,3,3,3,3,3,3,1,1,1,1,1,1,4,4,4,4,4,4];

withoutCorrection = [3,4]

correctedToneSequence = [1,3,1,4]
```

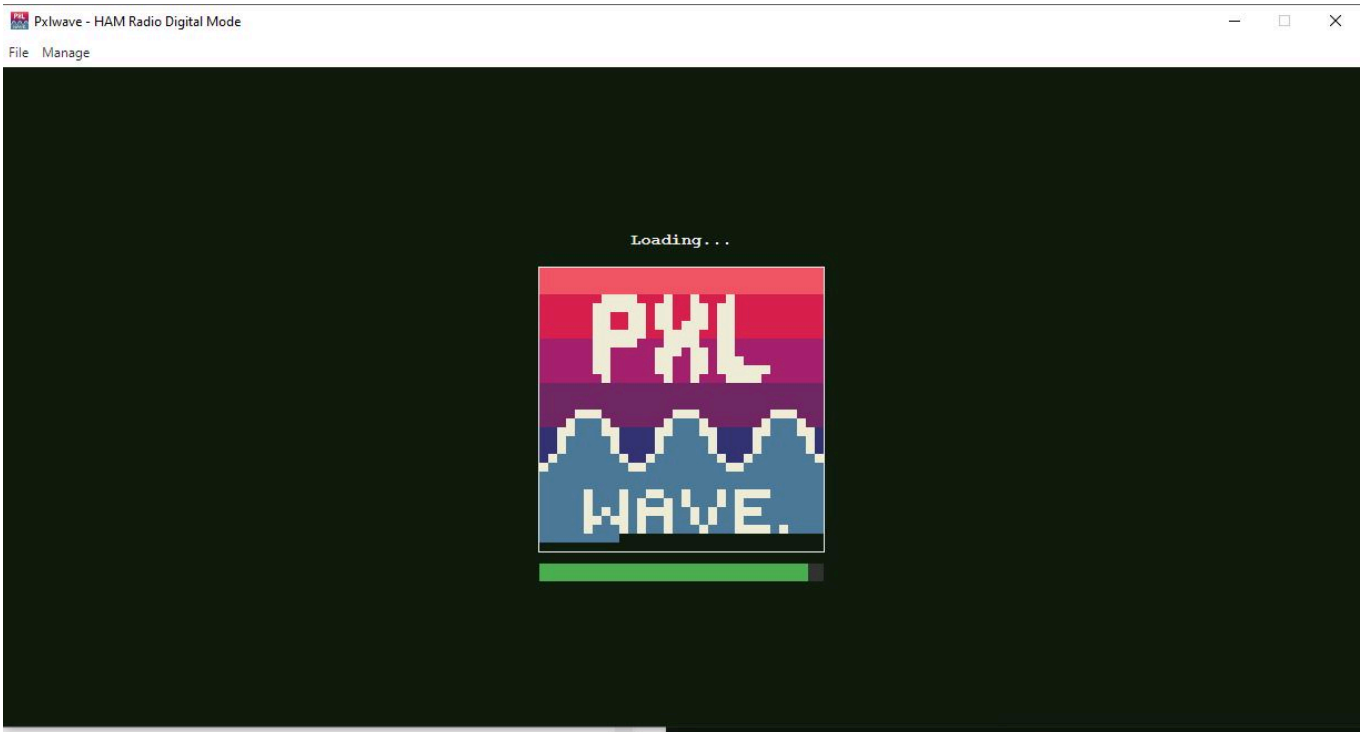
As you can see from above, while simple, this method has removed frequency oddities, and can connect strings of consecutive tones that don't quite meet the threshold to be accepted, lowering the error rate caused by doppler effect and drift.

RF Modulation

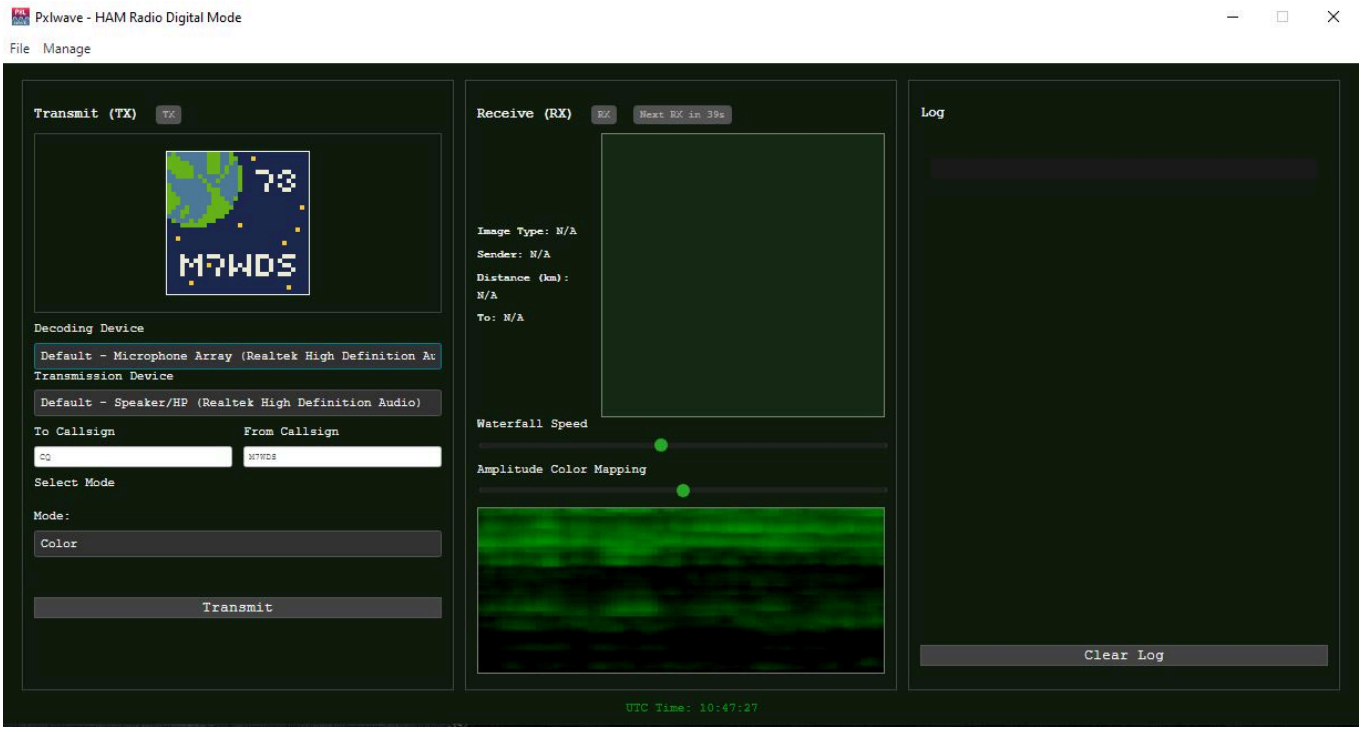
While the piQSL signal generation is modulated and demodulated by computer and the piQSL software, transmission over the radio is relatively modulation agnostic, as long as it is a voice mode.

piQSL is predominantly developed for QRP and DX, so naturally it is more likely to be used over the HF frequency bands where SSB is the norm for amateur radio use. In line with other digital mode and to create a default, it is recommended to use USB.

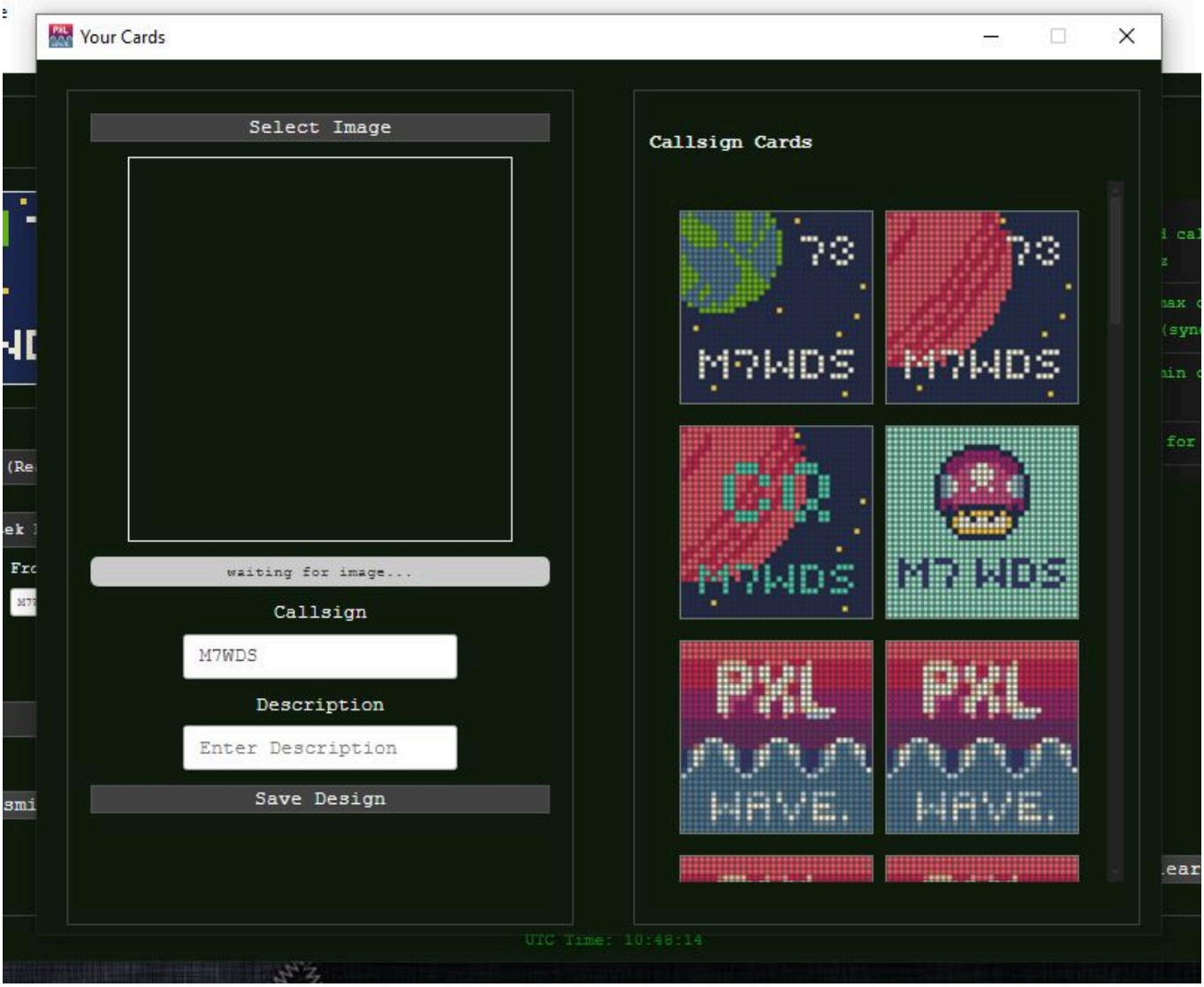
Early Screenshots



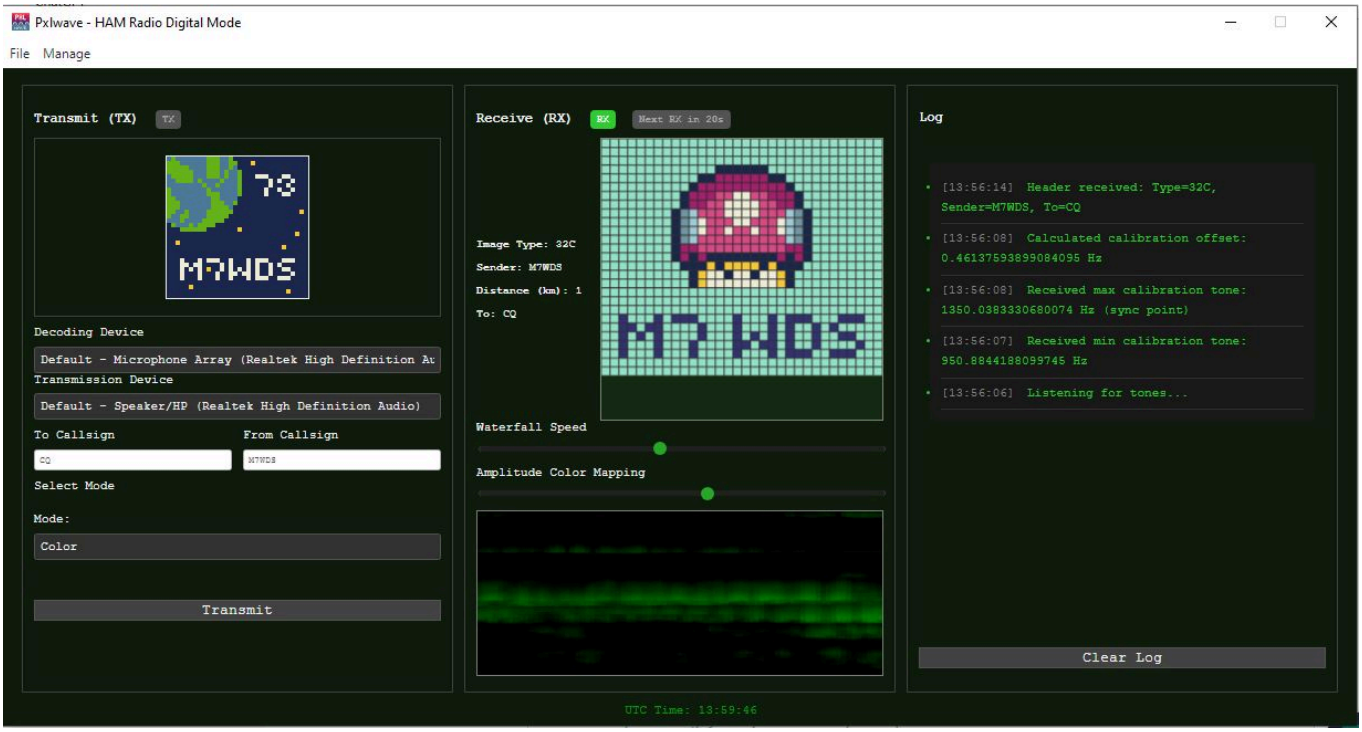
Loading Page



Main Window



Callsign cards; 73, CQ, and General QSL



Early Receive Example - missing lines 18 - 19 due to noise